Conception et développement d'un circuit multiprocesseurs en ASIC dédié à une caméra intelligente

Mohamed Amine Boussadi

N° d'ordre : D. U : 2552
EDSPIC : 690

UNIVERSITE BLAISE PASCAL - CLERMONT II

École Doctorale
Sciences Pour l'Ingénieur de Clermont-Ferrand

Thèse

présentée par :

Mohamed Amine Boussadi

pour obtenir le grade de

DOCTEUR D'UNIVERSITÉ

Spécialité : Vision pour la Robotique

Titre de la thèse :

Conception et développement d'un circuit multiprocesseurs en ASIC dédié à une caméra intelligente

Soutenue publiquement le : 25 février 2015 devant le jury :

M. Dominique Houzet	Rapporteur et Président
M. Gilles Sassatelli	Rapporteur
M. François Berry	Examinateur
M. Maxime Pelcat	Examinateur
M. Jean-Pierre Dérutin	Directeur de thèse
M. Alexis Landrault	Co-encadrant

À mes parents

Remerciements

Je remercie Jean-Pierre Dérutin, professeur de l'université Blaise Pascal, d'avoir accepté de diriger mes travaux de recherche. Je le remercie pour ses conseils, son encadrement et pour la confiance qu'il m'a accordée durant ces trois années de thèse pour mener à bien ces travaux de recherche.

Je remercie Alexis Landrault, maître de conférence de l'université Blaise Pascal, pour son encadrement sans faille et ses nombreux conseils qui m'ont permis d'avancer et mener à bien ces travaux de recherche.

Je remercie Thierry Tixier, ingénieur de recherche à l'Institut Pascal, de m'avoir transmis son expérience et ses connaissances. Grâce à son implication et ses conseils avisés, il a largement contribué au bon aboutissement de ce projet.

Je tiens également à remercier l'ensemble des membres de mon jury d'avoir accepté de juger et d'évaluer ces travaux de thèse. Je remercie Gilles Sassatelli, directeur de recherche CNRS au LIRMM, et Dominique Houzet, professeur à Grenoble-INP, et président du jury pour l'attention qu'ils ont accordée à la lecture de ce manuscrit ainsi que pour leur participation au jury en tant que rapporteurs. Je remercie François Berry, maître de conférence HDR de l'université Blaise Pascal, et Maxime Pelcat, maître de conférences de Institut National des Sciences Appliquées de Rennes, d'avoir pris de leur temps et d'avoir participés au jury en tant qu'examinateurs.

Enfin, je tiens à remercier toutes mes connaissances de m'avoir accompagnés et aidés pendant mes années d'études.

Résumé

Les capteurs intelligents de nos jours nécessitent des composants de traitement dotés d'une puissance suffisante pour exécuter les algorithmes à la cadence de ces capteurs d'images performants, tout en gardant une faible consommation d'énergie. Les systèmes monoprocesseur n'arrivent plus à satisfaire les exigences de ce domaine. Ainsi, grâce aux avancées technologiques et en s'appuyant sur de précédents travaux sur les machines parallèles, les systèmes multiprocesseurs sur puce (MPSoC) représentent une solution intéressante et prometteuse. Dans de précédents travaux à cette thèse, la cible technologique pour développer de tels systèmes était les FPGA. Or les résultats ont montré les limites de cette cible en terme de ressource matérielles et en terme de performance (vitesse notamment). Ce constat nous amène à changer de cible c'est-à-dire à passer sur cible ASIC nécessitant ainsi de retravailler profondément l'architecture et les IPs qui existaient autour de la méthode existante (appelée HNCP, pour Homogeneous Network of Communicating Processors).

Afin de bénéficier de la performance offerte par la cible ASIC, les systèmes multiprocesseurs proposés s'appuient sur la flexibilité de son architecture. Combinés à des squelettes de parallélisation facilitant la programmabilité de l'architecture, les circuits proposés permettent d'offrir des systèmes supportant le portage en temps réels de différentes classes d'algorithme de traitement d'images. Le résultat de ce travail a abouti à la fabrication d'un circuit intégré à base d'un seul processeur et de ses périphériques en technologie ST CMOS 65nm dont la surface est d'environ 1 mm^2 et à la définition de 2 architectures multiprocesseurs flexibles basées sur le concept des squelettes de parallélisation (une architecture de 16 cœurs de processeur en technologie ST CMOS 65 nm et une deuxième architecture de 64 cœurs de processeur en technologie ST CMOS FD-SOI 28 nm).

Mots-clés : Architecture parallèle, Traitement d'images en temps réel, Caméra intelligente, Système embarqué, MPSoC, ASIC, FPGA, 65nm CMOS, 28nm CMOS FD-SOI.

Abstract

Smart sensors today require processing components with sufficient power to run algorithms at the rate of these high-performance image sensors, while maintaining low power consumption. Monoprocessor systems are no longer able to meet the requirements of this field. Thus, thanks to technological advances and based on previous works on parallel computers, multiprocessor systems on chip (MPSoC) represent an interesting and promising solution. Previous works around this thesis have used FPGA as technological target. However, results have shown the limits of this target in terms of hardware resources and in terms of performance (speed in particular). This observation leads us to change the target from FPGA to ASIC. This migration requires deep rework at the architecture level. Particularly, existing IPs around the method (called HNCP for Homogeneous Network of Communicating Processors) have to be revisited.

To take advantage of the performance offered by the ASIC target, proposed multiprocessor systems are based on the flexibility of its architecture. Combined with parallel skeletons that ease programmability of the architecture, the proposed circuits allow to offer systems that support various real-time image processing algorithms. This work has led to the fabrication of an integrated circuit based on a single processor and its peripheral using ST CMOS 65nm technology with an area around 1 mm^2. Moreover, two flexible multiprocessor architectures based on the concept of parallel skeletons have been proposed (a 16 cores 65 nm CMOS multiprocessors and a 64 cores 28 nm FD-SOI CMOS multiprocessors).

Keywords : Parallel architecture, Real-time image processing, Smart camera, Embedded system, MPSoC, ASIC, FPGA, 65nm CMOS, 28nm CMOS FD-SOI.

Table des matières

Table des figures

Introduction

De par les progrès technologiques constants, l'évolution continue des flots de conception et l'amélioration régulière des techniques de fabrication, les circuits intégrés ont connu une évolution remarquable du point de vue de l'intégration et de son corollaire que constituent les performances. Cette évolution a permis notamment d'avoir des capteurs d'images performants et possédant une résolution très importante (plusieurs méga-pixels). Toutefois, ces caractéristiques rendent la tâche de traitement des données de plus en plus complexe au fur et à mesure que la quantité de données à traiter augmentait. Progressivement, les systèmes monoprocesseur se sont avérés inefficaces pour réaliser le traitement en temps réel des applications modernes. Considérant le domaine d'applications de l'Institut Pascal qui est la vision artificielle et afin d'assurer un traitement en temps réel, des solutions de traitement parallèle connue sous le nom "d'architecture parallèles" sont étudiées, développées et fabriquées.

De précédents travaux effectués au sein de l'Institut Pascal ont permis de mettre en place une méthodologie de prototypage rapide basée sur une architecture multiprocesseur homogène communicant appelé HNCP (Homogeneous Network of Communicating Processors). Cette méthodologie se propose de répondre à la fois aux défis matériels mais également logiciels issus de la complexité des applications du domaine du traitement d'images en temps réel. Car en plus de la conception matérielle, la complexité des applications embarquées rendent de plus en plus difficile la gestion de l'aspect logiciel des systèmes. Dans la méthode HNCP, le concept de squelettes de parallélisation (harnais de parallélisme récurrents) est exploité. Nativement, la méthodologie HNCP est basée sur une cible FPGA, flexible mais non optimale en terme de performances (ressources disponibles, consommation et vitesse). Cette situation nous amène à changer de cible technologique, c'est-à-dire à migrer vers une cible ASIC.

Cette thèse s'inscrit donc dans la poursuite des travaux préalablement menés sur la méthode HNCP et propose des évolutions architecturales et technologiques afin de faire face aux lacunes de la méthodologie. Cette évolution amène tout d'abord à repenser et/ou redévelopper des IP (Intellectual property) constituant l'architecture car la méthodologie HNCP repose sur deux flots de conception FPGA utilisant des outils et des IP propriétaires de chez Xilinx et Altera. Les nouvelles IP constituant l'architecture sont soit développées dans le cadre de ce travail depuis une spécification architecturale, soit choisies dans la communauté libre en apportant des améliorations et des adapta-

tions pour notre architecture. Ceci est un objectif important car il nous permet d'une part d'avoir un contrôle total sur l'architecture et d'autre part de contribuer aux projets libres de droit en vue d'une architecture multiprocesseur libre. Un autre point à améliorer dans la méthodologie HNCP est la topologie choisie actuellement. La topologie hypercube pose problème car au delà d'un certain nombre de processeurs, les liens entre processeurs deviennent trop longs ce qui affecte négativement le fonctionnement de l'architecture. Le nombre de liens par processeur entraîne également un problème de routage de l'architecture qui devient très difficile voir impossible sur certaines cibles technologiques.

Un des objectifs du travail présenté dans ce document est de proposer une architecture multiprocesseurs sur cible ASIC proposant une flexibilité maximale, dans la limite de la cible, en travaillant et en investiguant sur les aspects architecturaux et logiciels, le concept des squelettes de parallélisation restant l'axe fort de ce travail.
La première étape après la création des IPs de la nouvelle architecture consiste en la réalisation du premier circuit ASIC au sein de l'institut Pascal. Ce circuit intègre les IPs constituant la brique de base de notre futur multiprocesseur.
Une étude des technologies ASIC et des architectures envisagées a été faite afin de fabriquer ce premier circuit. Ce dernier permet de valider les IPs choisies pour l'architecture multiprocesseur tout en préservant un coût raisonnable pour un premier circuit. Au delà de la validation de la brique de base de notre futur multiprocesseur, la fabrication de ce premier circuit va mettre en place une solide base pour la conception des ASIC numériques au laboratoire. Ce travail présente également un intérêt certain pour les débouchés dans le monde académique.

A partir de la validation de cette première brique de base, l'objectif est de proposer deux circuits multiprocesseurs : un de 16 cœurs de processeur en technologie 65 nm et le deuxième de 64 cœurs de processeur en technologie 28 nm. Ces deux circuits sont basés sur le concept des squelettes de parallélisation et nécessitent des développements matériels et logiciels.

Le présent manuscrit est structuré ainsi :

- **le chapitre 1** décrit le contexte général de cette thèse en décrivant le domaine d'application s'y rattachant, puis la chaîne de traitement d'images en mettant le focus sur l'aspect traitement et les cibles technologiques. Nous nous intéressons ensuite aux précédents travaux réalisés au sein du laboratoire qui ont abouti à la méthodologie HNCP et nous expliquons la nécessité de passer au monde de l'ASIC,
- **le chapitre 2** revisite quelques notions de base sur les MPSoC (Multi-Processor System on Chip), puis un état de l'art des MPSoC les plus récents sur cible ASIC. Enfin, nous discutons le contexte de cette thèse vis-à-vis de cet état de l'art,
- **le chapitre 3** présente les développement matériels, afin de permettre le passage de la cible FPGA vers la cible ASIC,

- **le chapitre 4** présente le premier circuit réalisé au sein de l'institut Pascal. Nous détaillons tout d'abord le choix de la technologie, puis le choix de l'architecture, et enfin le flot de conception utilisé pour la fabrication du circuit et concluons sur les résultats,
- **le chapitre 5** présente l'architecture multiprocesseurs à 16 cœurs basée sur le concept des squelettes de parallélisation (SCM, FARM et PIPE) utilisés par la méthodologie HNCP et nous concluons sur les résultats.
- **le chapitre 6** introduit le troisième prototype réalisé dans le cadre de cette thèse. L'architecture est basée sur une extension et amélioration du deuxième prototype sur les aspects architecturaux et technologiques et nous concluons sur les résultats.

Chapitre 1

Le contexte de la thèse

1.1 Introduction

Dans ce chapitre, nous présentons le contexte général de cette thèse. Nous allons tout d'abord présenter notre domaine d'application qui est le traitement numérique des images en temps réel dans un contexte de recherche sur les véhicules autonomes et de la mobilité innovante développée à l'Institut Pascal. Ensuite, nous présentons la chaîne de traitement d'images en mettant le focus sur les caméras intelligentes et les différentes cibles technologiques permettant la réalisation matérielle du module de traitement constituant le cœur de cette thèse. Nous nous intéressons ensuite aux précédents travaux réalisés au sein de notre équipe (DREAM) de l'Institut Pascal qui ont abouti à la méthodologie de prototypage rapide appelée HNCP. Nous illustrons ensuite les limites de cette méthodologie et la nécessité de passer au monde de ASIC.

1.2 Architecture dédiée à la vision

La robotique mobile nécessite des systèmes avancés de perception artificielle. En effet, un robot mobile ayant une tâche de navigation autonome doit d'une part percevoir son environnement et d'autre part se localiser afin de s'asservir sur une trajectoire de référence correspondant à sa mission [1]. En conséquence, l'architecture matérielle doit être adaptée au domaine d'application dans lequel elle se situe. Dans celui de la vision, les images reçues doivent être traitées au même rythme que leur acquisition, c'est ce qu'on appelle l'exécution en temps réel. Si l'on considère le cas historique d'une vidéo standard à 25 images par seconde (la persistance rétinienne étant généralement prise égale à 1/25ème de seconde), le traitement d'une image ne doit pas dépasser 40 ms pour assurez un traitement en temps réel du système.
La notion de traitement en temps réel nous amène à parler des modules matériels et/ou logiciels utilisés pour le traitement, car ils déterminent si le système peut fonctionner en temps réel. Le traitement peut être réalisé soit par le biais de composants électroniques programmables ou de composants moins généralistes où la fonction de traitement est directement réalisée sur silicium pour plus d'efficacité. Mais avant de décrire ces modules

de traitement, il est nécessaire de donner un aperçu sur les composants qui nous permettent de recevoir et de transmettre les images au module de traitement, c'est-à-dire les capteurs d'images (caméras).

1.3 Les caméras intelligentes

1.3.1 Principe du capteur d'image

Par définition un capteur est un système qui permet de prélever une grandeur physique (température, pression, vitesse, etc ...) pour la transformer en grandeur électrique (tension, courant ou charge,etc ...). Un capteur intelligent est un capteur intégrant un interface d'acquisition, de traitement, et de communication. En conséquence, ce qui différencie les caméras intelligentes des caméras standards, c'est le fait de disposer au plus près du capteur (sous forme intégrée) de l'électronique permettant d'acquérir les images, de les stocker, de les traiter et aussi de communiquer avec d'autres systèmes. La notion de caméra intelligente sous-entend également la notion de contrainte d'encombrement. Dans certaines applications, l'usage d'un PC combiné au capteur n'est pas envisageable pour des raisons évidentes de place et dans ce cas, il faut développer une caméra intelligente satisfaisant le cahier des charges.

1.3.2 Caméras intelligentes dans la littérature

Les caméras intelligentes ont un large domaine d'application et ont fait l'objet d'études de la part de plusieurs acteurs universitaires et industriels notamment concernant l'aspect "exploration de nouvelles méthodes et architectures". Le tableau 1.1 présente un aperçu des plates-formes les plus courantes basées sur les caméras intelligentes. Nous pouvons constater que chacune de ces caméras a été développée pour une (ou des) application(s) spécifique(s).

1.3.3 Caméra intelligente utilisée dans le cadre de ces travaux

Une des caméra intelligente utilisée dans le cadre ces travaux de thèse est la DreamCam [9]. La DreamCam est une caméra intelligente modulaire. L'architecture de la caméra est réalisée avec cinq modules reliés entre eux comme le montre la figure ci-dessous (Fig. 1.1). Le module de traitement est à base de FPGA (Cyclone-III EP3C120 [10]) de chez Altera [11]. Le FPGA est associé à un dispositif de stockage qui est constitué de six blocs de mémoire RAM. Le module "capteur d'image" et le module "communication" peuvent être facilement remplacées ou mis à jour afin de modifier le type de l'imageur ou la couche de communication. Dans l'état actuel, il est possible d'utiliser deux liens de communication (USB 2.0 ou Giga-Ethernet) et deux capteurs d'image (MT9M031 (1,2 Mpx) ou EV76C560 (1,3 Mpx)).

Caméra	Capteur	CPU	Alimentation	Applications
ITI [2](2004)	LM-9618 CMOS	DSP TMS320C6415	secteur	Contrôle de la circulation
CMUcam [3](2007)	CMOS nivision	Processeur ARM7	batterie	robotique
MeshEye [4](2007)	ADNS-3060 optical mouse sensor + CMOS VGA	Microcontrolleur AT91SAM7S	batterie	imagerie distribué
SeeMOS [5](2007)	CMOS Cypress Lupa 4000	FPGA Stratix 60	secteur	suivi rapide de motifs
LE2I-Cam [6](2007)	CMOS Micron (MTM9M413)	FPGA Virtex II	secteur	imagerie à grande vitesse
WiCa [7](2007)	VGA CMOS	Xetal IC3D	batterie	détection de véhicule et estimation de la vitesse

TABLE 1.1 – Exemple de caméras intelligentes dans la littérature [8]

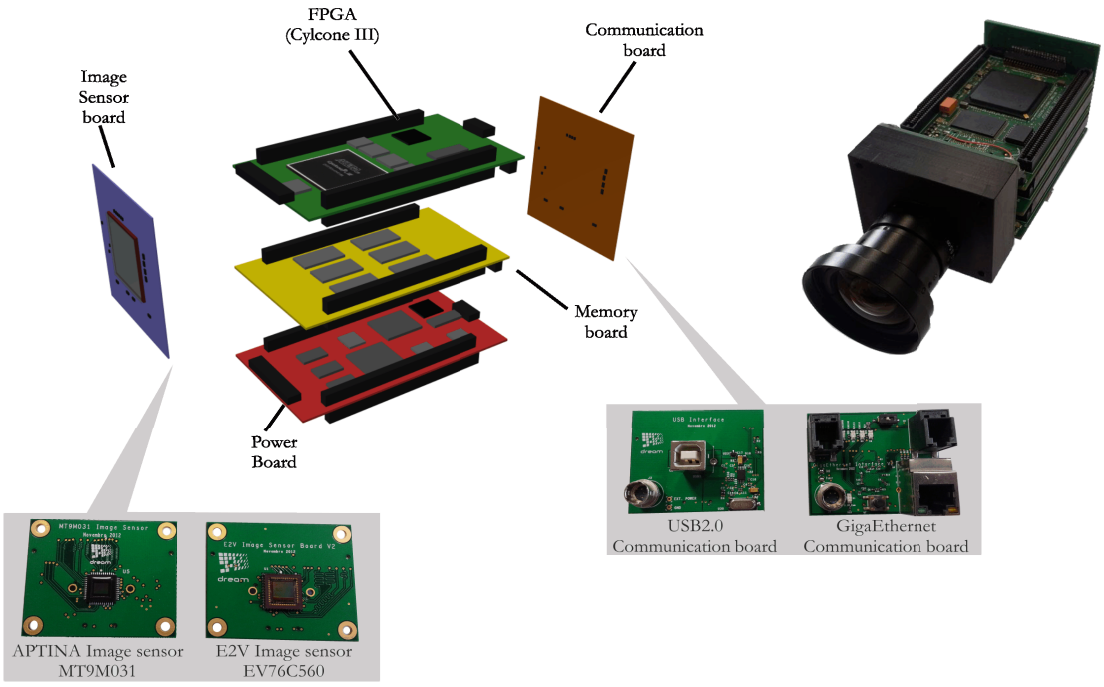


FIGURE 1.1 – Architecture de la DreamCam [9]

1.3.4 Impact de la haute résolution sur le traitement

Grâce aux progrès technologiques, de conception et avec l'évolution des techniques de fabrication, les capteurs d'images sont de plus en plus performants et possèdent une résolution très importante. Ces systèmes permettent de répondre aussi bien à des problématiques de vision industrielle qu'à des besoins de mesure et de supervision performants. Cette performance rend la tâche de traitement très difficile à cause de la grande quantité de données à traiter. La tâche est encore plus difficile lorsqu'on est dans le domaine de la vision artificielle, où le traitement d'images se fait en temps réel. Il est nécessaire dans ce cas, de faire appel à des solutions de traitement parallèle connues sous le nom des architectures parallèles. Dans le cadre de ce travail, les architectures parallèles dédiées à la vision seront plus particulièrement étudiées.

1.4 Architectures parallèles pour la vision artificielle

Dans le but de trouver l'implantation la plus optimale pour un algorithme ou une classe d'algorithmes donnée, les aspects logiciels (algorithmiques) et matériels (architecturaux) doivent être combinés et étudiés simultanément afin de trouver l'architecture adéquate en ciblant le maximum de performance. Pour ce qui des aspects logiciels, réduire le temps de la mise en œuvre de l'application est mis en avant. Pour ce qui est des aspect matériels, la réduction du coût matériel doit être mis en avant. Cette méthodologie est connue sous le nom de l'Adéquation Algorithme Architecture (AAA). Les études effectuées en AAA ont montré que le parallélisme ou les machines parallèles permettait de répondre efficacement à cette problématique. Un système ou machine parallèle est définie comme un ensemble d'unités de traitement (processeur par exemple) travaillant en parallèle et communicant. Nous distinguons selon la classification de Flynn [12] quatre principaux types de parallélisme (SISD, SIMD, MISD et MIMD). Cette classification est basée sur les notions de flot de contrôle (deux premières lettres de l'acronyme) et du flot de données (deux dernières lettres de l'acronyme).
Un exemple typique d'une machine SISD (Single Instruction Single Data) est la machine de Von Neuman. Dans cette machine, une seule instruction est exécutée et une seule donnée est traitée à tout instant. La machine SIMD (Single Instruction Multiple Data) est une machine qui peut exécuter une seule instruction sur plusieurs données en même temps. A l'inverse de celle-ci, la machine MISD (Multiple Instruction Single Data) peut exécuter plusieurs instructions en même temps sur la même donnée. Cela peut paraître paradoxal mais cela recouvre en fait un type très ordinaire de micro-parallélisme dans les micro-processeurs modernes (les processeurs vectoriels et les architectures pipelines). Enfin, la machine MIMD (Multiple Instruction Multiple Data) permet d'exécuter plusieurs instructions en même temps sur plusieurs données. Ce type de machine est très populaire et est celle retenue dans le cadre de ce travail notamment pour sa flexibilité et sa capacité à répondre à de nombreux problèmes (applications).

La figure suivante (fig. 1.2) montre les quatre types principaux de parallélisme selon la classification de Flynn :

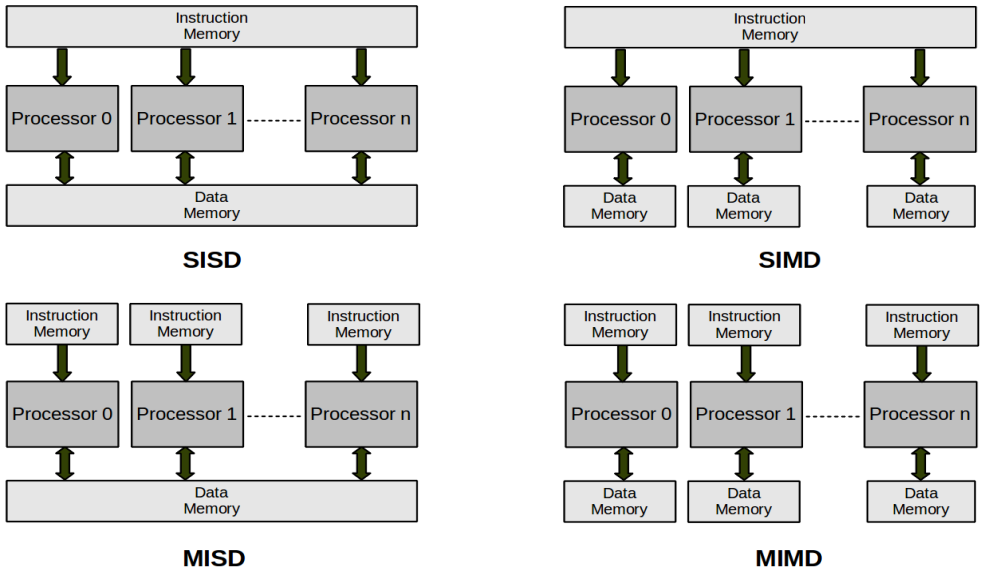


FIGURE 1.2 – La classification de Flynn

Nous pouvons distinguer deux types de machines MIMD : la machine MIMD-SM (Shared Memory) et la machine MIMD-MD (Memory Distributed).
La classe de machines à mémoire distribuée (MIMD-DM) est sans doute la plus performante bien que ce type de machine soit plus difficile à mettre en œuvre que les machines à mémoire partagée (MIMD-SM) notamment pour les aspects synchronisation et communication. Pour les systèmes à mémoire partagée, la distribution des données est totalement transparente pour l'utilisateur. Ce qui est très différent pour les systèmes à mémoire distribuée où l'utilisateur doit distribuer les données sur les processeurs ainsi que gérer l'échange de données entre processeurs. Donc, comme inconvénients : la communication entre processeurs est beaucoup plus lente dans les systèmes MIMD-DM. Toutefois, les avantages des systèmes MIMD-DM sont significatifs : le problème de la bande passante qui hante les systèmes de mémoire partagée est évité car la bande passante s'adapte automatiquement au nombre de processeurs.

Le gain de temps espéré (speedup), qui doit être idéalement égal au nombre de processeurs, est rarement atteint. En pratique, le parallélisme est difficile à contrôler, de plus il est difficile de faire une distinction tranchée entre le modèle à mémoire partagée et distribuée (dans la réalité, la situation est très hybride). Finalement, le modèle MIMD-MD a été le modèle choisi pour nos architectures depuis plusieurs années.

Après avoir abordé les modèles de parallélisme existant, la question de leur mise en œuvre doit être traitée. Une des solutions les plus couramment utilisée depuis une dizaine d'années est de mettre en œuvre un système sur puce. Dans la section suivante sont donc abordés deux types de systèmes sur puces utilisés dans le cadre de cette thèse.

1.5 Les systèmes sur puce

La conception de systèmes embarqués de traitement d'image amène à développer des systèmes complets intégrant des parties logicielles et matérielles, de la mémoire, des microprocesseurs (pouvant être spécialisés), des périphériques d'interface ou tout autre composants. Aujourd'hui, ces systèmes résultants sont de manière quasi-systématique amenés à être intégrés sur une seule puce d'où l'appellation de systèmes sur puce (SoC pour System on Chip). Les systèmes dédiés au traitement d'images embarqué nécessitent un grand nombre de composants (capteurs, processeurs, mémoires, périphériques de communication) afin de réaliser la chaîne de traitement et satisfaire les applications visées. Les premiers systèmes électroniques étaient essentiellement des circuits réalisés sur carte. L'assemblage se fait sur des circuits imprimés (PCB (Printed Circuit Board)) à partir de composants électroniques posés sur des pastilles et reliés électriquement entre eux via des pistes.
Le principal critère de choix d'une carte électronique est le nombre de couches constituant la carte. Les cartes basées sur une seule couche sont de plus en plus rares dans les domaines à courant faible et à forte intégration. Les cartes les plus répandues de nos jours sont les cartes multicouches car elles permettent pour un même encombrement d'intégrer plus de fonctions électroniques. Aujourd'hui, les circuits imprimés multicouches sont présents dans presque tous les domaines y compris le domaine du traitement d'images. Un circuit multicouches peut avoir jusqu'à 48 couches (ce nombre est encore en augmentation) selon le domaine d'application et la technologie utilisée.
Certes, les circuits sur carte multicouches ont permis d'augmenter le nombre de composants et de réduire la surface occupée, mais ce type de conception ne remplace pas les systèmes sur puce ; elle est complémentaire. Ces systèmes ont été mis en place afin effectivement de répondre en partie à l'évolution exponentielle des systèmes sur puce, en particulier le nombre de broches qui ont augmenté considérablement à cause de la forte intégration des puces (par exemple, les FPGA Stratix V de chez Altera on jusqu'à 1120 entrées/sorties). Ce point implique de parler de l'évolution des systèmes sur puce.

Le nombre de transistors intégrés au maximum dans une puce a doublé tous les 18 mois, c'était la prédiction de Gordon Moore publiée en 1965 dans son article *"Cramming More Components onto Integrated Circuits"* [13]. A partir de cette date, la densité d'intégration des transistors sur une même puce n'a cessé d'augmenter. La figure suivante (fig. 1.3) montre l'évolution du nombre de transistors dans les processeurs de 1971 jusqu'à 2020 (les systèmes à base de processeurs sont parmi les circuits les plus complexes en terme de surface dans le domaine des systèmes sur puce numérique).

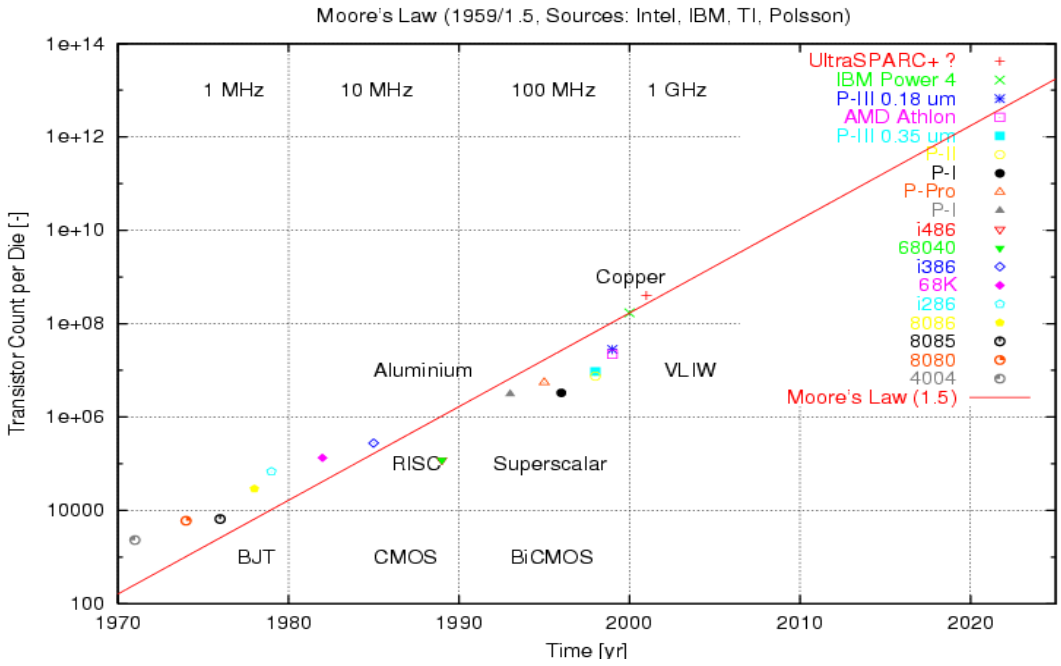


FIGURE 1.3 – Évolution du nombre de transistors dans les processeurs [14]

Cette évolution remarquable est due principalement à l'avancement des procédés technologiques dans le domaine de la fabrication des puces. Cette évolution a touché toute la chaîne (conception, fabrication et test). La figure 1.4 montre les principales étapes dans la chaîne de fabrication des systèmes sur puce.

Comme pour toute conception de système, la première étape est la spécification de l'architecture qui permet de réduire de manière évidente le temps de conception et ainsi le temps de mise sur le marché (TTM pour Time To Market), ce qui réduit aussi le coût du système. La conception se fait à l'aide de logiciels de CAO (Conception Assistée par Ordinateur) nommée également en anglais EDA (pour Electronic Design Automation). Le résultat de la conception est la création des dessins des masques (ou layout) qui, sous un format de fichiers appelé GDSII (Graphic Database System), est transmis à la fonderie. La phase de conception est terminée laissant place à la phase de fabrication. La première étape dans la phase de fabrication est la création des masques technologiques. Le processus de fabrication est basé sur l'utilisation d'un procédé photolithographique complexe pour réaliser le masque de chaque couche. Toutes les étapes de fabrication se font dans des salles blanches. Après la création des masques viens l'étape d'implantation sur le wafer (une séries d'étapes : photolithographie, lithographie électronique, électrochimie, métallisation,..). Une fois les étapes d'implantation réalisées, le wafer est prêt à être découpé. Chaque puce élémentaire est montée sur un support, soudée et enfin encapsulée.

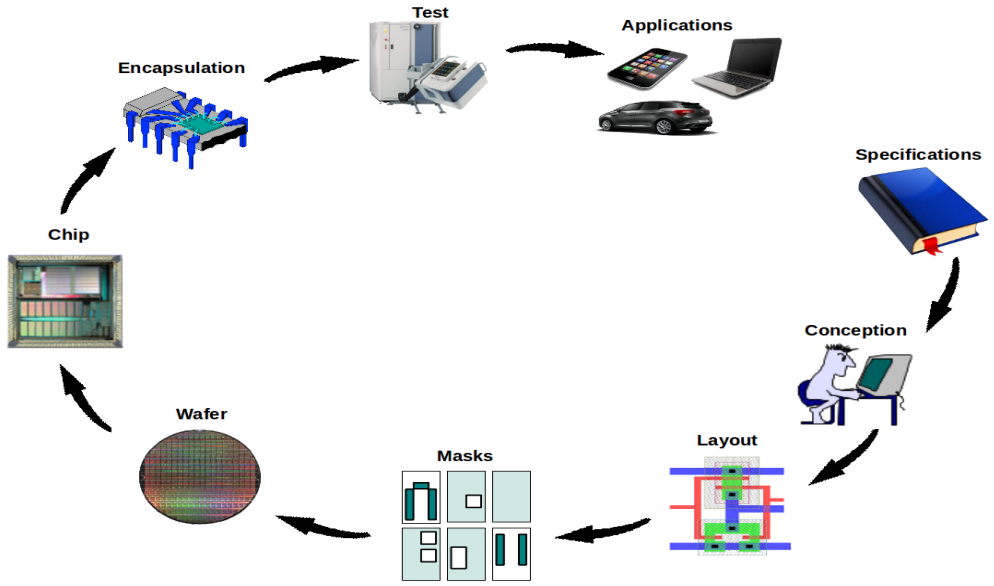


FIGURE 1.4 – Chaîne de fabrication des systèmes sur puce

1.6 Les cibles technologiques

Le domaine des circuits électroniques intégrés a connu des progrès considérables depuis 30 ans. Dès les premières années, les recherches dans le domaine architectural on permit d'améliorer les performances des circuits intégrés en augmentant la fréquence de fonctionnement, réduisant la surface de silicium occupée et en réduisant aussi la puissance consommée. Ces avancées architecturales ont amené à créer des classes de circuits intégrés. Globalement les circuits intégrés sont classés en deux grandes classes : les circuits standards et les circuits sur mesures. La figure suivante (1.5) illustre la classification des circuits intégrés :

Parmi ces familles de circuits intégrés (fig. 1.5), nous nous intéressons à la catégorie des circuits sur mesure (custom) et dans cette catégorie, il existe deux types de circuits dans la catégorie des circuits à moitié sur-mesure (semi-custom) que sont les circuits à logique programmable et les circuits à base de cellules standards.
Ils sont nommés ainsi (semi-custom) car ils comportent des éléments architecturaux prédéfinis (à dimensionner sur mesure en fonction de l'application), ce qui permet un gain de temps de conception comparé aux approches "hand layout". Dans le cadre de cette thèse, nous utilisons les circuits de type FPGAs à des fins de prototypage, avant d'entamer la phase de fabrication du circuit ASIC (Application Specific Integrated Circuit) à base de cellule standard.

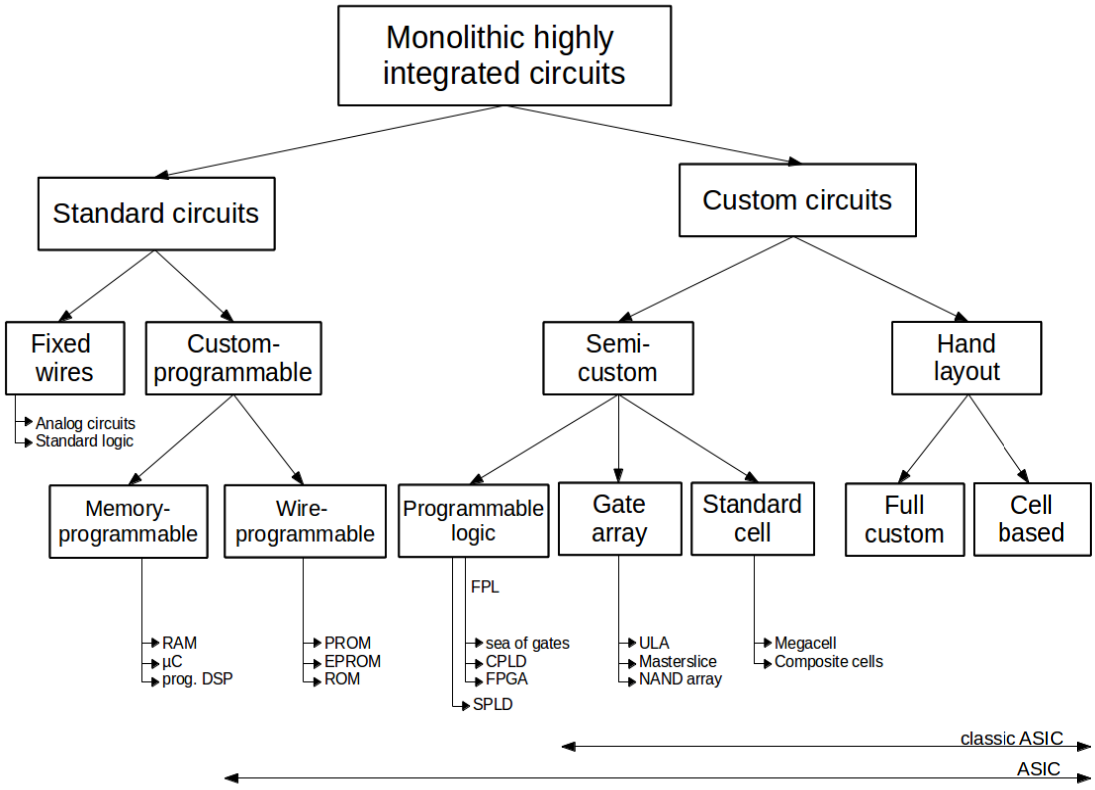


FIGURE 1.5 – Classification des circuits [15]

1.6.1 Les FPGAs

Les FPGA (Field Programmable Gate Arrays) ou « réseaux logiques programmables » sont des circuits intégrées re-configurables. L'avantage de ce type de circuit est sa grande souplesse qui permet de les réutiliser autant de fois que l'on désire. Une fois le circuit hors tension, l'architecture implantée est perdue et le circuit redevient "vierge". Grâce à l'évolution technologique, les FPGA sont de plus en plus rapides et à plus haute intégration. Par exemple le Stratix V (5SEE9H40C4N) fabriqué en technologie 28 nm de chez Altera contient 317000 LABs/CLBs, 840000 éléments logiques/cellules, 64210944 bits de mémoire RAM et 696 broches d'entrées/sorties, ce qui permet d'implanter des applications de complexité importante.
Les circuits FPGA sont constitués d'une matrice de blocs logiques entourés de blocs d'entrées/sorties. L'ensemble est relié par un réseau d'interconnexion. Ces trois composantes sont programmables. La figure 1.6 montre l'architecture interne d'un FPGA :

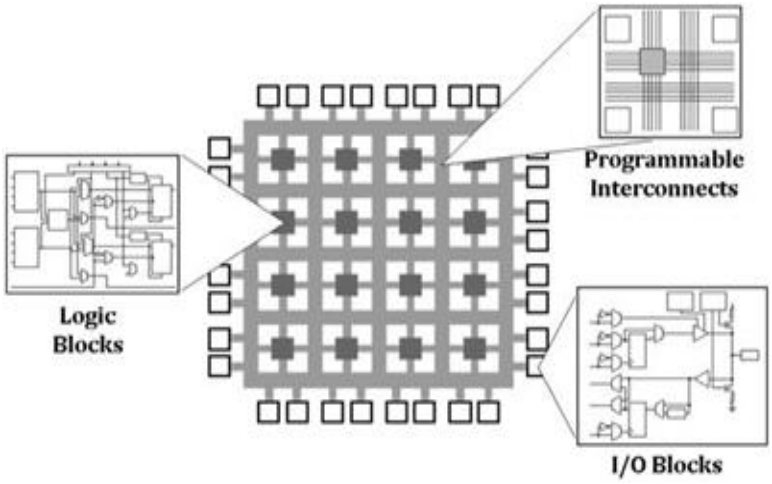


FIGURE 1.6 – Structure interne d'un composant FPGA

1.6.2 Les ASICs

Les ASICs (Application Specific Integrated Circuit) comme il a été vu précédemment est une grande famille (les FPGA sont classés comme des ASICs), mais généralement ce terme est utilisé pour décrire la famille des circuits complètement sur mesure (full custom) ou les circuit à base de cellules standards (Standard cell). Pour la suite de ce manuscrit, lorsque nous utilisons le terme ASIC, nous faisons allusion au circuit semi sur mesure à base de cellules standard. Les ASICs à base de cellules standards sont les circuits les plus répandus dans la catégorie des circuit numériques car ils offrent les meilleures performances (vitesse, puissance, surface) tout en préservant des coûts intéressants (pour les grands volumes) et un bon TTM. La richesse des cellules utilisées dépend de la technologie et du fournisseur de cette technologie.

La figure 1.7 montre les principaux éléments de base d'un circuit à base de cellules standards. L'architecture est constituée d'une partie active basée sur deux types de cellule. Les cellules standard comme les portes combinatoires et séquentielles (INV, NAND, OR, Bascule D, Flip-Flop,...) et les macros qui sont des éléments spéciaux comme les mémoires (RAM, ROM, ...) ou encore les générateurs d'horloge comme des PLL. L'architecture contient aussi des cellules d'entrées/sorties (IO pads). Nous pouvons distinguer deux types d'IOs. Les IOs dédiées à l'alimentation du circuit et les IOs dédiées au transfert de données. La taille des IOs est généralement supérieure, de l'ordre de 50 μm x 100 μm, cette taille facilite le packaging. Pour ce qui est du packaging, plusieurs méthodes sont utilisées pour relier les IOs. La méthode utilisée dans la figure 1.7 s'appelle collage de fil (Wire bonding). Il reste deux surfaces à définir : le "seal ring" qui est une marge ajouter par les fondeurs pour éviter des erreurs lors de la coupure de la puce ("die") et la cavité qui est une zone vide qui sépare le "seal ring" du package.

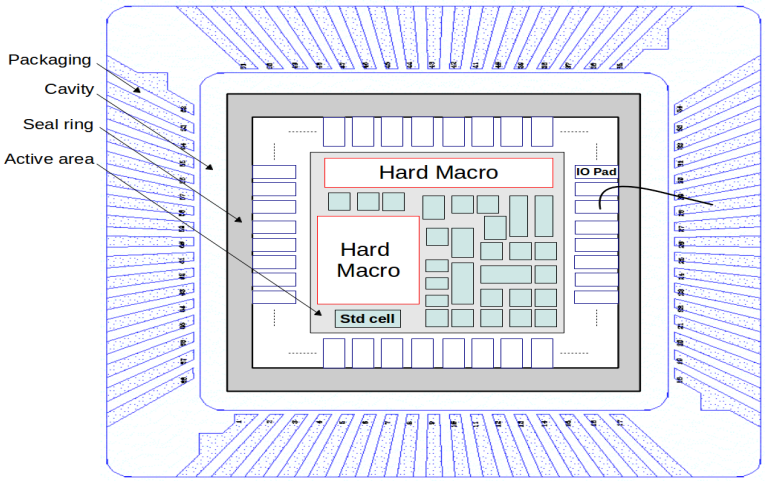


FIGURE 1.7 – Circuit ASIC à base de cellules standard et de macros

1.6.3 Flot de conception sur FPGA et sur ASIC

Le coté gauche de la figure 1.8 présente le flot de conception sur FPGA (le flot présenté est celui des FPGAs de chez Altera, mais il est très similaire aux autres flots FPGA). Deux langages se sont imposés pour la description matérielle : le Verilog et le VHDL. Pour la description logicielle, le langage C et ses dérivés comme le C++ se sont majoritairement imposés. D'autres langages de description de haut niveau tel que le SystemC peuvent être utilisés. Ce langage est utilisé par plusieurs concepteurs et flots d'aide à la spécification, exploration et conception rapide d'architecture (tel que dans l'ESL SCE (Electronic System-Level System-on-Chip Environment [16])). Une fois l'architecture décrite, elle est transcrite au niveau RTL afin de générer l'architecture finale.

Du point de vue FPGA, une fois le RTL généré, il doit être synthétisé : concrètement le synthétiseur va traduire le code HDL en un fichier nommé "netlist" décrivant le circuit comme un assemblage de blocs logiques. L'étape de synthèse permet de donner un aperçu sur le nombre de blocs logiques utilisés pour réaliser l'architecture du système. L'étape suivante est le placement. Dans cette étape, le nombre exact de blocs logiques utilisés dans l'implantation est connu. Une fois le système placé, il va être router. Enfin, pour implanter le système sur la cible, l'outil va créer un fichier de programmation appelé "bitstream" qui contient toutes les spécifications de configuration du circuit programmable. Si le système implanté contient une partie logicielle dans son architecture (par exemple un processeur), la partie logicielle est fusionnée avec la partie matérielle avant l'implantation sur cible.

L'architecture décrite par les concepteurs doit être simulée pour en valider le bon fonctionnement. Elle peut s'effectuer à différents niveaux. Les trois simulation qui sont un passage obligatoire pour chaque conception sont : la simulation comportementale

qui s'exécute sur une description HDL du système, la simulation après synthèse qui s'exécute sur le résultat de la synthèse et enfin la simulation après placement-routage qui s'exécute sur le résultat du placement-routage. Dans les deux dernières simulations (après synthèse et après placement-routage), les aspects temporels sont pris en compte.

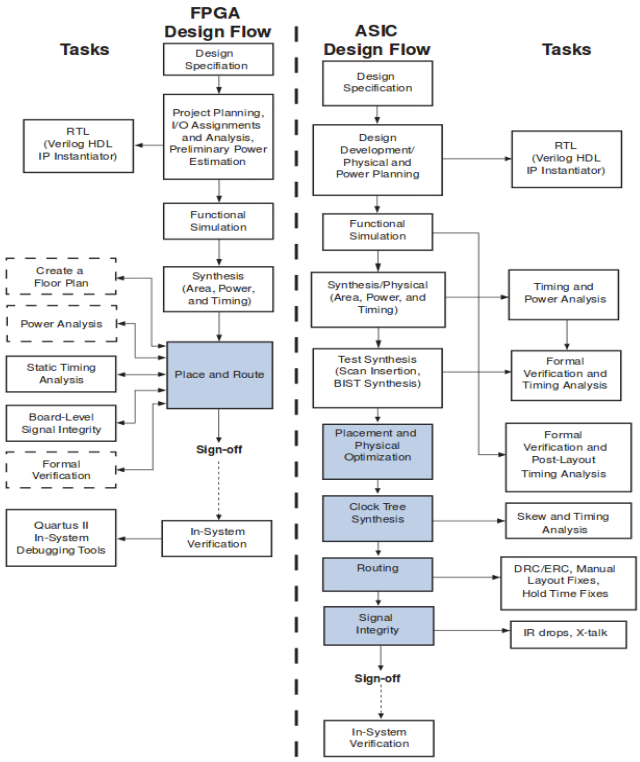


FIGURE 1.8 – Comparaison entre un flot de conception ASIC et un flot de conception FPGA[17]

Pour ce qui est du flot de conception ASIC (voir figure 1.8, partie de droite), il est globalement similaire au flot FPGA (description, synthèse et placement routage), mais il faut procéder aux différentes étapes manuellement (le flot n'est pas totalement automatisé, même si chaque étape individuellement fait appel à un outil CAO), ce qui peut rendre le flot long et parfois compliquer à dérouler. Une autre différence est qu'il faut utiliser une technologie cible (process de fabrication) à spécifier (plus connu sous le nom de Design Kit, fichier intégrant l'ensemble des règles de dessins et les cellules standards). Le développement d'un circuit ASIC numérique nécessite tout d'abord une description HDL. Une fois le code écrit, la conception est réalisée à l'aide de logiciels dédiés pour la CAO, accompagnés de logiciels de test. La première étape après l'écriture du code est la synthèse. Cette étape permet de générer une netlist généralement en verilog qui décrit l'architecture dans la technologie choisie, ce qui n'est pas le cas pour les FPGA (la technologie est celle du FPGA). La seconde étape est le placement-routage du circuit. Cette étape est la plus longue et est délicate à faire pour le flot

ASIC. Des allers-retours entre les différentes étapes du flot peuvent être faites selon les résultats de simulation obtenus. Théoriquement, les deux flots sont très similaires, mais le flot FPGA est plus simple et plus automatique à mettre en œuvre notamment car les composants sont physiquement disponibles et fonctionnels (déjà fondus sur silicium). L'outil va juste choisir la bonne configuration selon les critères d'optimisation choisis par le concepteur (surface, vitesse ou puissance). Par contre dans l'approche ASIC, le circuit doit être créé physiquement à partir des cellules standards. Une différence capitale par rapport au flot FPGA est qu'une erreur commise lors de la conception peut s'avérer très coûteuse car irréversible une fois le circuit fondu.

1.7 FPGA vs ASIC

Les deux cibles technologies présentées précédemment ont pour but la réalisation d'une fonctionnalité électronique (traitement, communication,...) via un circuit intégré, mais le choix d'utiliser la technologie FPGA ou la technologie ASIC dépend des besoins et des objectifs (notamment en terme de performances, de coût et de TTM) de l'utilisateur. La comparaison entre les deux cibles peut se faire en trois points : performances, coût et flexibilité :

La performance : Les trois principaux paramètres de performance d'un circuit intégré sont la vitesse, la surface et la consommation. La figure 1.9 montre l'écart de performance entre les FPGA et les ASICs.

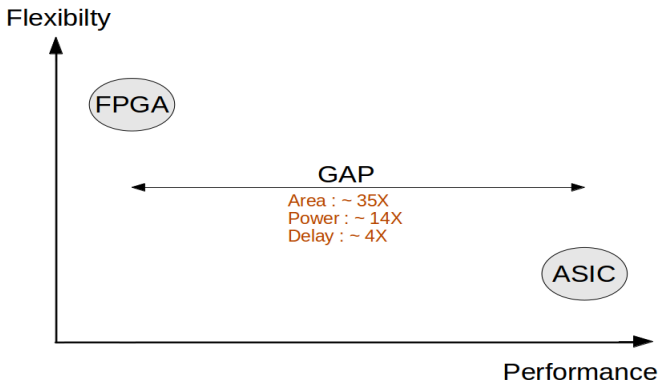


FIGURE 1.9 – FPGA vs ASIC

En terme de vitesse, une implantation sur FPGA est entre 3,4 à 4,6 fois plus lente, en moyenne, qu'une implantation sur ASIC à base de cellules standard [18] à technologie identique. Pour ce qui est de la consommation : En moyenne, un FPGA consomme 14 fois plus qu'un ASIC (consommation dynamique) [18]. La surface pour une implantation sur FPGA en utilisant des LUTs comme éléments logiques de base est d'environ 35 fois plus grande qu'une implantation sur ASIC à base de cellules standard [18].

Il faut noter aussi que la surface occupée par une implantation donnée dépend aussi du type d'éléments utilisés dans l'architecture à implanter. Le tableau suivant (tab. 1.2) montre l'impact de quelques éléments de base sur la surface en fonction de la cible technologique (FPGA Vs ASIC) [19].

Surface ocuppée	ASIC	FPGA
Mémoires	élevée	moyenne
Multiplieurs	élevée	moyenne
Registres	moyenne	faible
Additionneurs	moyenne	faible
Multiplexeurs	faible	élevée

TABLE 1.2 – Surface occupée par quelques éléments de base

Le coût : La conception d'un système sur ASIC est très coûteuse, comparée à celle sur un FPGA pour de faibles volumes à cause du côut du "ticket d'entrée" (coût fixe). Cet handicap peut être évité si la quantité de production est élevé. La figure suivante (Fig. 1.10) montre le coût par rapport au volume de production. Pour un volume de production supérieur à 1000 pièces pour les ASICs structurés et supérieur à 5000 pièces pour les ASICs à base de cellule standard, l'approche ASIC devient plus intéressant que l'approche FPGA.

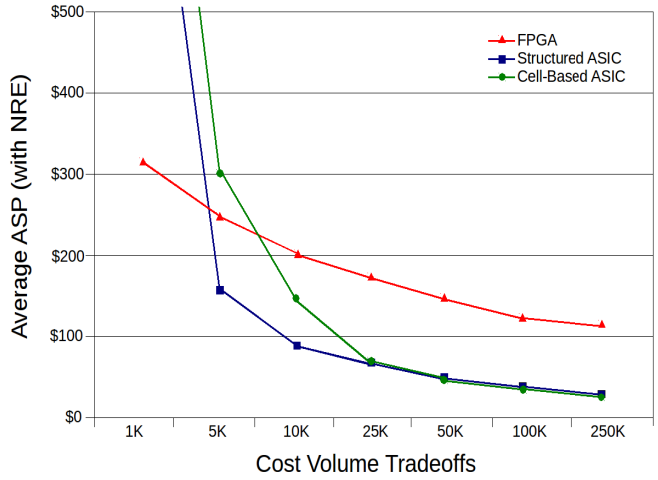


FIGURE 1.10 – Coût par rapport au volume de production[[20]

La flexibilité : Le FPGA offre une grande souplesse de configurabilité d'un point de vue matérielle. L' ASIC par contre ne l'est pas : une fois le circuit fabriqué, l'architecture est figée et la seule fonctionnalité qu'il peut réaliser est celle décrite par la spécification mis en place.

1.8 Solutions de traitement pour la complexité croissante des applications

Pour de nombreuses applications, la complexité a atteint un point où les exigences de performance ne peuvent plus être assurées par les architectures standard basées sur un seul processeur. C'est pourquoi de nombreux systèmes ont tendance à se baser sur une architecture multiprocesseur (par exemple, la méthodologie HNCP que nous détaillons dans la section suivante). Il existe d'autres approches tel que la logique câblée avec une implantation matérielle de l'application en HDL. Par exemple nous pouvons citer l'implantation de l'algorithme de détection de points d'intérêt en VHDL sur cible FPGA [8]. Il est évidant que cette approche est la plus performante pour cette application, par contre le temps de conception la pénalise. Pour résoudre ce problème des approches pour la génération rapide de code matériel (Verilog ou VHDL) est utilisée. Par exemple nous pouvons citer CAPH [21] qui est un langage dédié à la synthèse d'applications flot de données sur FPGA. Une autre approche très utilisées dans le domaine des applications graphiques est basée sur l'utilisation des GPU (Graphics Processing Unit) qui sont devenus des outils puissants pour le calcul intensif parallèle [22]. Toutefois, leur programmation reste un point difficile. Dans le cadre de cette thèse nous détaillons la méthodologie HNCP qui combine le prototypage rapide basé sur les squelettes de parallélisation sur cible FPGA.

1.9 HNCP : Une méthodologie de prototypage rapide

Dans cette section, nous présentons une méthodologie de prototypage rapide pour des réseaux multiprocesseurs s'appelant HNCP (Homogeneous Network of Communicating Processors) [1] en introduisant les aspects logiciels et matériels, ainsi que l'outil utilisé pour la génération de l'architecture. Nous abordons ensuite les squelettes de parallélisation qui sont le cœur logiciel de la méthodologie. Enfin, nous discutons des limites de cette méthodologies et des solutions proposées.

1.9.1 Présentation de la méthodologie

Comme son nom l'indique, la méthodologie est basée sur le concept de réseau de processeurs homogènes communicants. La figure 1.11 montre une représentation graphique du flot de conception par HNCP. Le centre de ce flot de conception est l'outil Cubegen pour FPGA. Cet outil, permet de générer une architecture multiprocesseurs de topologie hypercube. La méthodologie utilise des outils fournis par le fabricant de FPGA choisi lesquels permettent la synthèse et l'implantation de l'architecture. Dans un flot de conception basé sur les outils Xilinx (comme c'est le cas dans la fig. 1.11, la description de l'architecture matérielle est définie dans le fichier (*.mhs). Cette architecture est synthétisée puis combinée avec la partie logicielle basée sur la programmation parallèle de l'algorithme en utilisant des fonctions de communications spécifiques basées sur le concept des squelettes de parallélisation (SCM, FARM et PIPE que nous

décrivons dans la section 1.9.3). L'ensemble sera finalement implanté sur la cible définie précédemment par l'utilisateur.

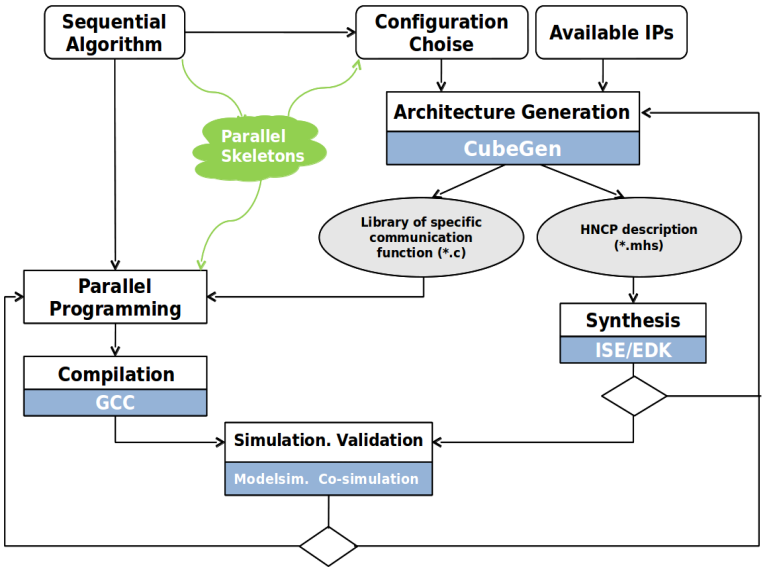


FIGURE 1.11 – Représentation graphique du flot de conception par HNCP

1.9.2 Aspect matériel de la méthodologie

La méthodologie propose une architecture multiprocesseurs homogène. Le choix de topologie est l'hypercube. Pour un degré d'hypercube n, on a 2^n processeurs et n liens par processeur. Chaque nœud de l'architecture est constitué d'un processeur, d'une unité de mémoire, d'une unité de communication et d'une unité de gestion du flot vidéo. La figure 1.12 montre une architecture cubique à 8 processeurs.

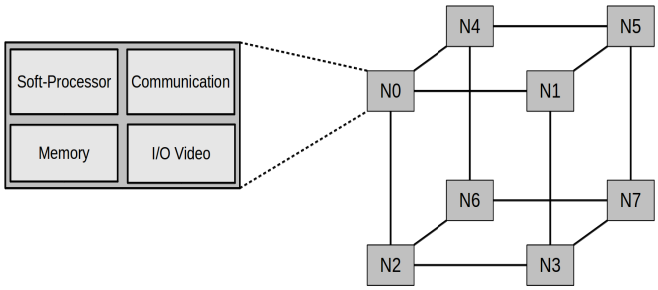


FIGURE 1.12 – Réseau hypercube de 8 processeurs

1.9.2.1 Unité de calcul (processeur)

Les deux cibles FPGA supportées par la méthodologie sont Xilinx ou Altera. Le processeur fourni par Altera s'appelle le NIOS [23] et celui fourni par Xilinx s'appelle le MicroBlaze [24]. Les deux processeurs sont des softcores 32 bit à jeu d'instructions réduit RISC (Reduced Instruction Set Computer), ayant une architecture HARVARD. Avec une telle architecture, les instructions et les données stockées dans la mémoire sur puce (on chip) du processeur sont accessibles simultanément en un cycle d'horloge. Disposant d'une architecture configurable, une implantation softcore peut être reconfigurée contrairement à un processeur dit hardcore dont le cœur dispose de sa propre puce sur le FPGA qui ne peut être modifiée (on dit alors que le processeur est réalisé en "dur"). La figure ci-dessous (fig.1.13) montre l'architecture interne du softcore MicroBlaze :

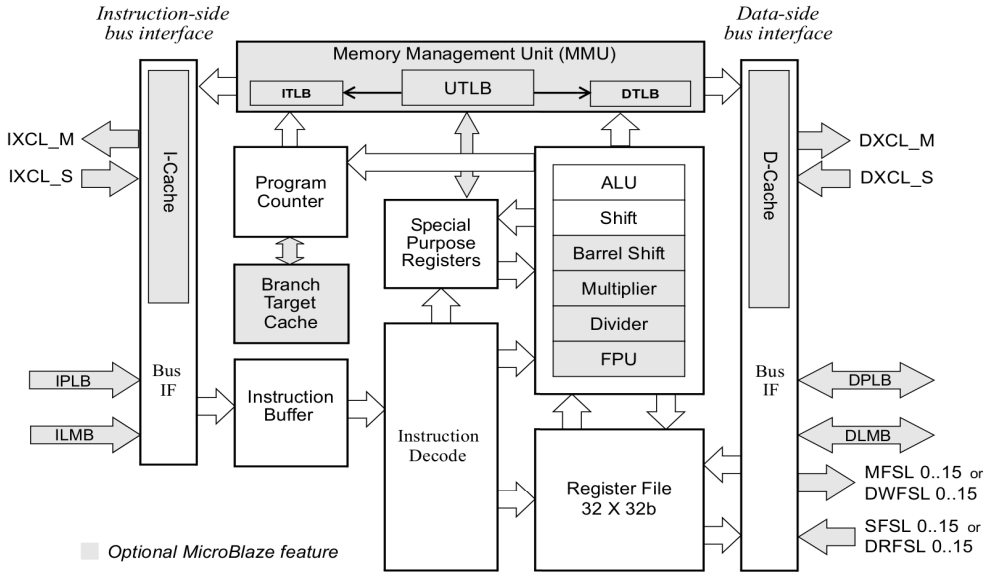


FIGURE 1.13 – Architecture interne du MicroBlaze [24]

1.9.2.2 Unité de communication

La méthodologie HNCP propose deux modes de communication :

– La communication point à point : cette communication est réalisée via des FIFOs telles que les bus FSL (Fast Simplex Link) de Xilinx. Le bus FSL est un moyen rapide de communication entre le processeur et une autre entité. Chaque lien FSL est unidirectionnel, point-à-point et met en œuvre une FIFO avec une certaine profondeur (à définir par l'utilisateur) afin de stocker plus de données et ne pas bloquer le processeur lors de l'écriture sur le bus. Le bus utilise des signaux de contrôle pour indiquer l'état de la FIFO (pleine, vide ...) et l'opération à effectuer (lecture ou écriture). Le bus utilise aussi un mode de communication asynchrone.

Cette option de synchronisation n'a pas été utilisée par la méthodologie HNCP à l'origine mais elle est utilisée dans le cadre de cette thèse.

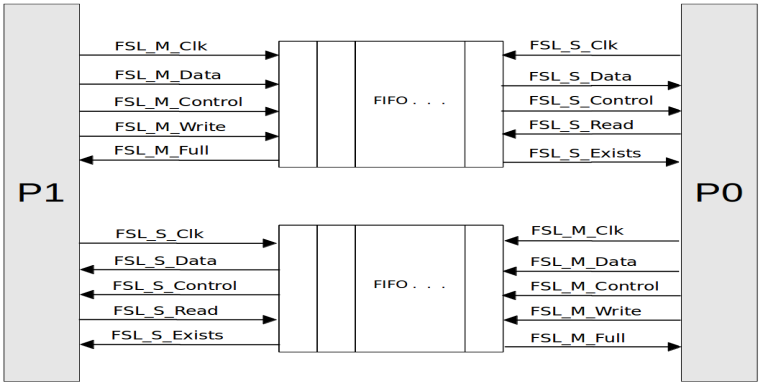


FIGURE 1.14 – Réseau de 2 processeurs avec interface FSL

– La communication par routage : cette communication est réalisée à l'aide d'un DMA (Direct memory access) et d'un routeur développés dans le cadre de travaux précédents [25]. Le DMA-Routeur dispose d'un nombre de ports adaptables aux nombres de nœuds. De plus, certaines caractéristiques du routeur peuvent être adaptées en fonction des besoins en communication (taille des buffers, taille des paquets). La figure suivante montre un réseau de 4 processeurs MicroBlaze avec routeur et interface DMA.

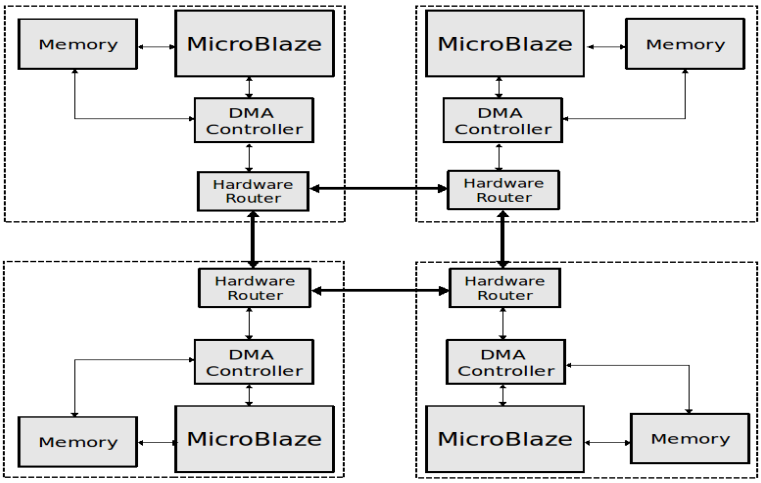


FIGURE 1.15 – Réseau de 4 processeurs avec routeur et interface DMA

1.9.2.3 Unité de mémoire

La taille de la mémoire programme (mémoire locale) dépend de la taille du programme à implanter et des données à traiter. Pour un réseau de processeurs communicants par des liens FSL, une seule mémoire est utilisée. Pour un réseau de processeurs communicants par DMA-Router, deux autres modules mémoires sont nécessaires, un pour la réception (mémoire de réception) et un autre pour l'envoi (mémoire d'émission). Tous les mémoires utilisées sont à double port. Pour la mémoire programme, un port est utilisé pour les données et l'autre pour les instructions. Pour les mémoires réception et émission, un port est utilisé par le module DMA, l'autre port est utilisé par le processeur.

1.9.2.4 Unité de gestion du flot vidéo

L'unité de gestion du flot vidéo permet au processeur d'accéder à l'image en temps réel. Ce module reçoit l'image depuis le capteur d'images et écrit l'image dans une mémoire (quatrième mémoire) accessible par le processeur. La gestion du flot vidéo est introduite théoriquement dans la méthodologie HNCP, elle n'a pas été mise en œuvre avant les travaux présentés ici.

1.9.3 Aspect logiciel de la méthodologie

Afin de mettre en œuvre un algorithme parallèle sur une architecture, des travaux antérieurs [26] ont montré qu'un ensemble commun et récurrent de squelettes [1] parallèles (modèles) correspondant au parallélisme statique de données, au parallélisme dynamique de données et au parallélisme de tâche peut être utilisé. L'utilisation de squelettes implique la nécessité de définir une bibliothèque paramétrable de communication. Dans la méthodologie HNCP, cette étape est complètement transparente pour l'utilisateur, car elle est générée automatiquement par l'outil selon l'architecture choisie. Dans cette section sont décrits les squelettes mis en œuvre par la méthodologie.

1.9.3.1 Le squelette SCM (Split, Compute and Merge)

Ce squelette est consacré au traitement de données régulier (parallélisme de données statique). Le même traitement est appliqué à chacune des données, ce qui entraîne une charge de travail très régulière. L'ensemble des données d'entrée est divisé (étape de Split) en un sous-ensemble de données, et chaque bloc est traité de façon indépendante sur chaque processeur (étape de Compute). Le résultat final est obtenu par la fusion

1. Le concept de squelette algorithmique repose sur le fait que l'on retrouve fréquemment les formes récurrentes que sont le parallélisme de données, de tâches et de flux dans les applications que l'on veux paralléliser.
En pratique un squelette algorithmique est défini comme un schéma parallèle générique, paramétré par une liste de fonctions qu'il est possible d'instancier et de composer. Les fonctions sont spécifiées par l'utilisateur et chaque squelette encapsule les communications qui lui sont propres.
Le concepteur n'a donc plus à gérer explicitement des tâches de communication ou de synchronisation "bas-niveau" et peut donc se concentrer sur des aspects plus algorithmiques de l'implantation [25].

du calcul de chaque sous-ensemble de données (étape de Merge). Le SCM est défini comme statique car son système de communication est entièrement connu à la compilation (fig. 1.16).

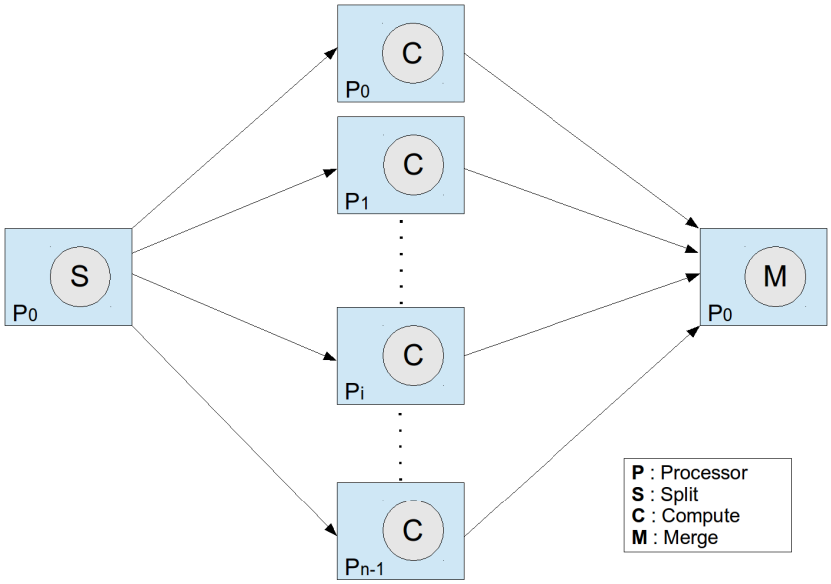


FIGURE 1.16 – Représentation graphique du squelette SCM

La figure 1.17 montre le mécanisme du squelette SCM tel qu'il a été défini dans la méthodologie HNCP. Le problème de l'implantation revient à placer et ordonnancer efficacement le graphe SCM sur le graphe correspondant à la topologie de processeurs retenue. Les trois fonctions de bases du squelette ("Split", "Compute" et "Merge") sont implantés dans tous les processeurs sachant que la fonction "Compute" est la même dans tout le réseau. Selon la position du processeur dans l'architecture, les fonctions "Split" et "Merge" sont différentes d'un processeurs à l'autre. Le principe de la fonction "Split" est de récupérer les données d'entrées. Toutefois, une fonction "Split" doit être capable de récupérer ses propres données d'entrée mais également de transmettre les données d'entrée aux processeurs voisins. Le principe de la fonction "Merge" est de transmettre les résultats : elle doit être capable de transmettre ces propres résultats et récupérer les résultats des processeurs voisins. Les données d'entrées et les résultats passent toujours par le processeur 0 comme le montre la figure 1.17, car ce dernier est le seul processeur à posséder un accès aux interfaces d'entrées/sorties, ce qui confère au processeur 0 un rôle particulier.

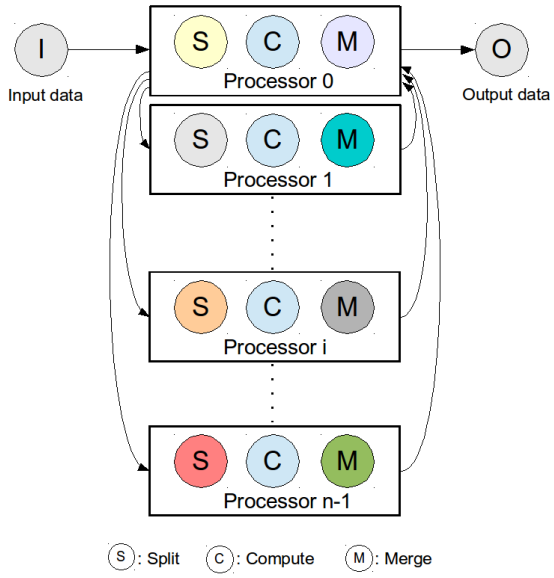


FIGURE 1.17 – Implantation générique du squelette SCM

1.9.3.2 Le squelette FARM

Dans certains algorithmes de vision artificielle, le traitement nécessite un ensemble de données irrégulières (parallélisme de données dynamique). Par exemple, une liste arbitraire de fenêtres de tailles différentes changeant à chaque itération d'image. Dans ce cas, une répartition statique n'est pas efficace en raison d'une charge de travail irrégulière entre les processeurs (fig. 1.18).

La figure 1.19 montre le principe du squelette FARM tel qu'il a été défini dans la méthodologie HNCP. Dans le cas de charges de travail irrégulières, un processeur appelé le fermier (d'où le nom de FARM) distribue directement les données ou les tâches aux processeurs travailleurs. Les tâches/données représentent un sous-ensemble de données (par exemple des zones d'intérêt dans l'image). Typiquement, le fermier commence par l'envoi (Send) d'une tâches/données à chaque travailleur. Ensuite, il attend (Receive) un résultat d'un travailleur et envoie immédiatement une autre tâche/donnée afin qu'il l'exécute (Compute). Ceci est répété jusqu'à ce qu'il ne reste aucune tâche/donnée à traiter et que les travailleurs ne sont plus en cours de traitement. De leur côté, chaque travailleur attend simplement une tâche/donnée, la traite et renvoie le résultat au fermier jusqu'à ce qu'il reçoit une condition d'arrêt du fermier. Cette technique permet d'équilibrer les charges de travail sur l'ensemble des processeurs.

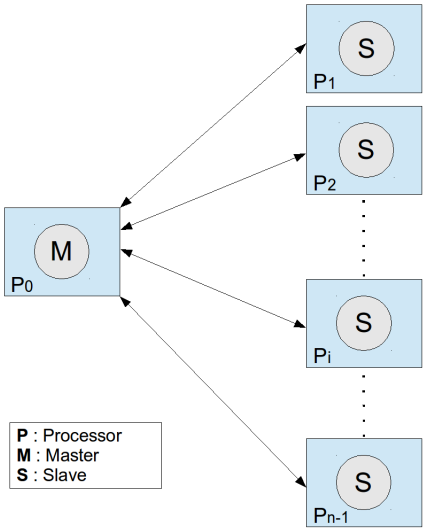


FIGURE 1.18 – Représentation graphique du squelette FARM

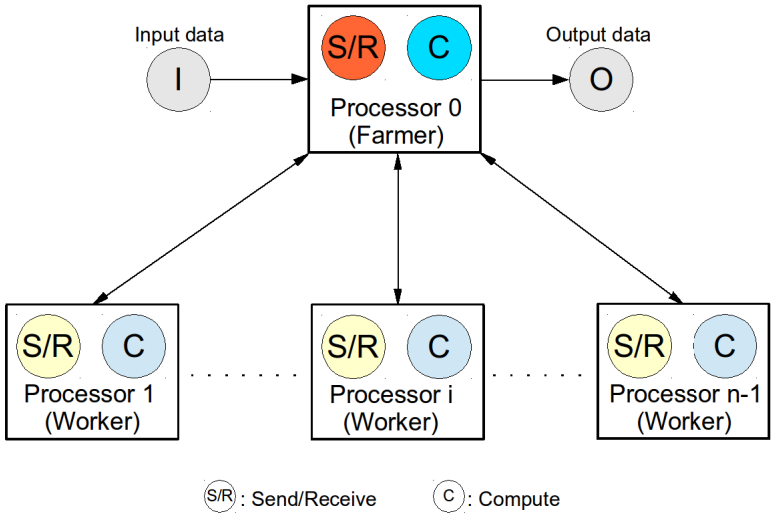


FIGURE 1.19 – Représentation graphique du squelette FARM

1.9.3.3 Le squelette PIPE

La figure 1.20 montre le principe du squelette PIPE. L'idée de base du squelette pipeline consiste à diviser le traitement en une série d'étapes successives, avec une synchronisation à la fin de chaque étape (parallélisme de tâches sur un flot). Ceci permet d'avoir un temps d'exécution qui varie au rythme de l'étape la plus lente, ce qui est beaucoup plus rapide que le temps nécessaire pour effectuer toutes les étapes de traitement à la fois. Le squelette PIPE permet de faire du parallélisme de tâches.

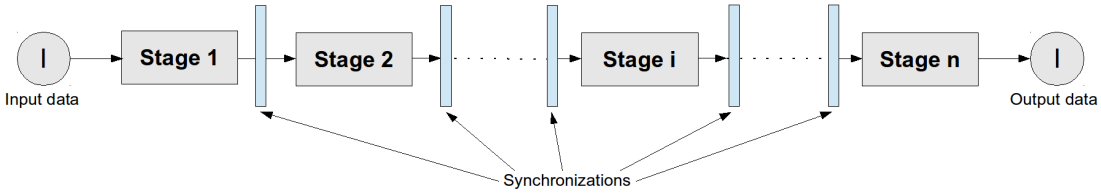


FIGURE 1.20 – Représentation graphique du squelette PIPE

1.9.4 L'outil CubeGen

Cet outil permet de définir les caractéristiques matérielles du système sur trois niveaux. Le premier niveau définit la topologie, le nombre de nœuds et le nombre d'étages de l'architecture. Le second niveau correspond à l'étape de choix des différentes IPs du nœud (softcore, module de communication ...). Enfin, le troisième niveau, les paramètres internes des composants sont spécifiés. CubeGen génère alors les fichiers d'entrée de l'environnement de développement EDK (Framework de chez Xilinx) afin de synthétiser, placer, router et enfin implanter l'architecture sur la cible et les bibliothèques de communication. La figure 1.21 montre un exemple de l'interface graphique de l'outil CubeGen. Cette fenêtre est la première étape de spécification, elle correspond au choix du nom et de l'emplacement du projet, elle permet aussi de choisir la carte et l'environnement de développement. La dimension et le nombre d'étage de l'hypercube sont sélectionnés dans la deuxième fenêtre. Pour une dimension n donnée, on aura 2^n processeurs. Par exemple, pour une dimension $n = 2$ et un nombre d'étage $m = 2$ on aura deux hypercubes de 4 processeurs chacun.

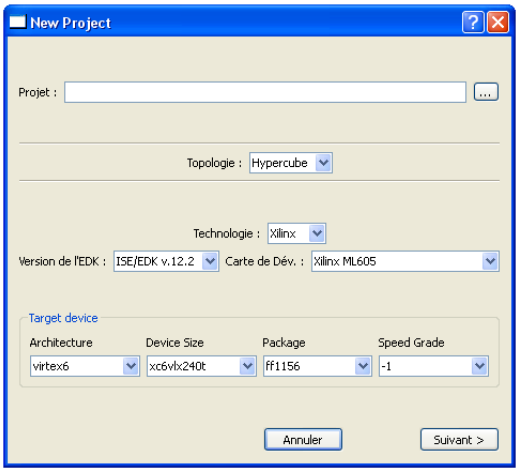


FIGURE 1.21 – Interface graphique de l'outil CubeGen

1.9.5 Les limites de l'architecture et de l'approche FPGA

La méthodologie HNCP a démontré l'efficacité des réseaux homogènes communicants [1] notamment le temps de mise en œuvre, ce qui a permis de faire le prototypage rapide et la parallélisation efficace de plusieurs algorithmes [27]. Néanmoins, cette approche possède des lacunes en terme de performances (surface, consommation et vitesse). Ces lacunes sont dues à deux principales causes : la topologie de l'architecture et la cible matérielle. Pour la première cause, dans une topologie hypercube au delà d'un certain nombre de processeurs, certains liens deviennent physiquement trop long, ce qui rend difficile le maintien des caractéristiques temporelles ("timing" dans l'étape du "backend") sur la cible. Le nombre de liens par processeur pose également problème dans ce type de topologie, ce qui rend le routage de l'architecture impossible sur les cibles technologiques (FPGA ou ASIC). En ce qui concerne la cible choisie, si les FPGA ont certes permis d'atteindre un niveaux de flexibilité très important surtout dans le développement d'applications numériques innovantes, leur utilisation devient bloquante pour des applications complexes pour des raisons de ressources disponibles, de consommation et de vitesse (voir la section FPGA vs ASIC).

1.10 Passage de FPGA vers ASIC

Le tableau 1.3 récapitule les architectures et les algorithmes implantés dans les deux précédentes thèses dans le cadre de ce projet (L. Sieler [1] et H.Chenini [27]). La dernière architecture implantée via la méthodologie HNCP est une architecture de type pipeline avec un total de 24 processeurs. Les pourcentages des ressources utilisées par celle-ci par rapport aux ressources totales disponibles sur la cible (Stratix IV : EP4SGX530) sont de 21% de slice et 79% de BRAM. Même si le FPGA utilisé dispose d'un grand nombre de ressources, il s'avère que la quasi intégralité de la surface du circuit est utilisée (surtout en terme de mémoire dans le cas de l'utilisation du squelette FARM). Deux autres inconvénients sont la consommation et la vitesse. Comme il a été montré dans la section (FPGA vs ASIC), les FPGA consomment 35 fois plus et ils sont 10 fois moins rapide qu'un ASIC dans la même technologie. Cette situation nous amène à changer de cible technologique, c'est-à-dire passer à une cible ASIC.

1.11 Conclusion

Dans ce chapitre, nous avons situé le contexte de cette thèse. Au travers de cette étude, il apparaît clairement l'importance du parallélisme afin de répondre aux besoins croissants des applications innovantes notamment dans le domaine de la vision. Il apparaît également clairement la nécessité de passer d'une cible FPGA vers une cible ASIC pour des raisons de ressources matérielles et de performances. Toutefois, cette solution, si elle permet d'augmenter la performance des architectures proposées, risque de diminuer la flexibilité. Un des objectifs du travail présenté dans ce document est de proposer une architecture multiprocesseurs sur cible ASIC proposant une flexibilité

maximale, dans la limite de la cible, en travaillant et en investiguant sur l'aspect architectural. Ceci introduit le chapitre suivant qui traitera des solutions multiprocesseurs et notamment celles dans le domaine circuits multiprocesseurs sur ASIC.

Année	Architectures et Algorithmes	Temps, efficacité et taille d'image	Résultats d'implantations
2010 [1]	**16** processeurs en SCM exécutant l'algorithme d'Harris et Stephen	9,72 ms 0,81 256×192 px	34,46% en Slice et 69,56% en BRAM (XC6VLX240T)
2010 [1]	**8** processeurs en FARM exécutant l'algorithme de Suivi de lignes de marquage au sol en contexte autoroutier	4,55 ms 0,52 128×196 px	56% en Slice et 93% en BRAM (XC6VLX240T)
2011 [1]	**4** processeurs en SCM exécutant l'algorithme d'Harris suivi de **8** processeurs en FARM exécutant l'algorithme de mise en correspondance de primitives	605 ms 0,39 256×192 px	62% en Slice et 60% en BRAM (XC6VLX240T)
2012 [27]	**16** processeurs en FARM exécutant l'algorithme de détection de visage (filtre à particules)	121 ms 0,42 256×192 px	38% en Slice et 97% en BRAM (XC6VLX240T)
2013 [27]	**32** processeurs en SCComCM (Split, Compute, all-to-all Communication, Compute, Merge) exécutant l'algorithme d'aide au stationnement d'un robot mobile par réseau de neurones	6,7 ms 0,40 256×192 px	20% en Slice et 51% en BRAM (EP4SGX530)
2014 [27]	**12** processeurs en SCM exécutant l'algorithme de mise en correspondance suivi de **12** processeurs en SCM exécutant l'algorithme d'estimation de pose (RANSAC)	247 ms 0,59 6600x2048 px	21% en Slice et 79% en BRAM (EP4SGX530)

TABLE 1.3 – Résultats décrits dans les thèses [1] et [27]

Chapitre 2

Les systèmes multiprocesseurs intégrés sur puce

2.1 Introduction

Dans le précédent chapitre, la méthode de prototypage rapide HNCP a été présentée. Nous avons également présenté les limites de cette approche en terme de performances notamment dues au à la cible choisie (FPGA). Nous avons aussi montré que le propos du présent travail est de dépasser ces limites en migrant vers une cible ASIC. Dans ce chapitre, nous allons définir quelques notions de base sur les multiprocesseurs avant de faire un état de l'art sur les projets multiprocesseurs actuels sur cible ASIC. Nous concluons ce chapitre en situant le contexte de cette thèse vis-à-vis de ces projets.

2.2 Les MPSoC (Multi-Processor System-on-Chip)

Un système multiprocesseurs est un système complexe à spécifier, à concevoir puis à réaliser, qui demande un nombre important de portes logiques en fonction du nombre de processeurs, de leur interconnexion, de présence ou non de mémoire, etc... pour une mise en œuvre sur silicium. Mais, dès que l'évolution des technologies d'intégration sur silicium a rendu possible la réalisation de tels systèmes, un grand nombre de solutions ont vu le jour. Toutefois, une partie de la théorie sous-jacente à la réalisation de MPSoC était déjà établie puisque les notions de parallélisme avaient déjà été étudié et mis-en-œuvre dans les systèmes sur carte [28] [29] [30]. Les travaux sur les MPSoC sont une manière de réétudier et de réinvestir les concepts liés à ces travaux sur les machines parallèles. Les MPSoC représentent une solution intéressante pour répondre à la demande croissante pour le calcul haute performance intégré dans plusieurs secteurs de marché tels que le multimédia, les communications, l'aérospatiale, l'imagerie médicale et la robotique. Les architectures multiprocesseurs régulières sont connues [30] pour être efficace dans le domaine d'application de traitement d'image, parce que ces applications sont très gourmandes en calculs et présentent souvent un haut degré de parallélisme de données.

2.3 Les topologies

La topologie d'un système multiprocesseurs est une caractéristique importante car elle influe directement sur la communication entre les processeurs. Elle permet de définir l'architecture du réseau de processeurs en établissant les connexions entre processeurs et éventuellement une hiérarchie entre eux. Il existe deux catégories de réseaux : les réseaux statiques qui sont constituées de liaisons point-à-point fixes entre les processeurs et les réseaux dynamiques qui ont des éléments de communication actifs, tels que des routeurs, qui peuvent changer le modèle de connexion selon le protocole. Dans un FPGA, le réseau peut être reconfiguré dynamiquement pour s'adapter aux modes de communication en utilisant la reconfigurabilité inhérente au FPGA [31].

La topologie peut être vue comme un graphe établissant l'organisation des différents nœuds de l'architecture. Selon les applications visées, une topologie peut être plus souhaitable qu'une autre en terme de performances (vitesse, congestions éventuelles et surface notamment). Le choix d'une topologie va également agir directement sur les communications entre nœuds et sur l'extensibilité du réseau. Effectivement, certaines topologies sont bridées en terme d'extensibilité. Par exemple la topologie complètement connectée devient difficilement extensible au delà d'un certain nombre de nœuds. Le nombre de liens augmentant très rapidement, la réalisation matérielle devient impossible. La figure 2.1 montre des exemples de topologies. Chaque topologie peut être vue comme un graphe constitué d'arêtes (lien de communication) et de sommets (nœud intégrant un processeur).

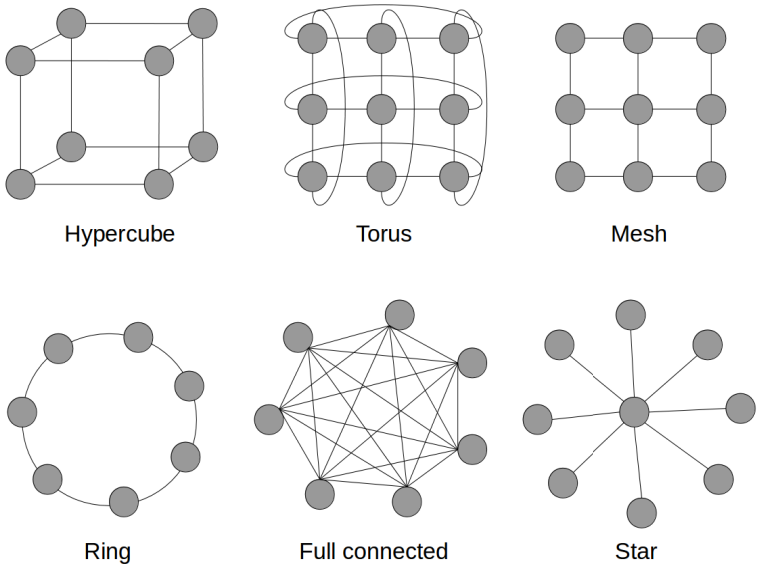


FIGURE 2.1 – Exemples de topologies

Les topologies présentées précédemment dans la figure 2.1 sont des topologies de base dans le domaine des architectures multiprocesseurs [32]. Ces topologies peuvent être caractérisées avec des notions issues de la théorie des graphes telles que :

- **Le degré du nœud** : représente le nombre de liens qui relient chaque nœud à ses voisins.
- **La régularité d'un réseau** : si le degré est constant, on dit que le réseau est à degré régulier.
- **Le diamètre du réseau** : représente la distance maximale séparant deux nœuds du réseau.
- **Le nombre de liens** : représente le nombre de liens maximum que peut avoir un nœud.
- **La bissection** : représente le nombre minimum de liens qu'il faut enlever pour partitionner équitablement un réseau en deux sous-réseaux.

Le tableau 2.1 résume les principales caractéristiques des topologies présentées dans la figure 2.1. La variable n correspond au nombre de nœuds constituant le réseau.

Topologie	Diamètre	Nombre de liens	Degré	Régularité du réseau	Bissection
Anneau	$\frac{n}{2}$	n	2	oui	2
Étoile	2	$n-1$	$1, n-1$	non	$\frac{n}{2}$
Grille	$2(\sqrt{n}-1)$	$2(n-\sqrt{n})$	$2, 3, 4$	non	$\sqrt{n}$
Tore	$\sqrt{n}$	$2n$	4	oui	$2\sqrt{n}$
Hypercube	$\log_2 n$	$\frac{n \log_2 n}{2}$	$\log_2 n$	oui	$\frac{n}{2}$
Complètement connecté	1	$\frac{n(n-1)}{2}$	$n-1$	oui	$\frac{n^2}{4}$

TABLE 2.1 – Caractéristiques d'un réseau selon la topologie

2.4 La hiérarchie de la mémoire

La mémoire est l'un des points les plus importants caractérisant un système multiprocesseurs. Un des points essentiels lors de la définition de la mémoire est son organisation, c'est-à-dire comment la mémoire est organisée sur l'ensemble des unités de traitement afin que chaque unité soit capable à tout moment d'accéder aux instructions et aux données sauvegarder dans cette mémoire. Dans une architecture multiprocesseurs, deux types d'organisation de mémoire (fig. 2.2) sont utilisées et vont être présentées dans les paragraphes suivants.

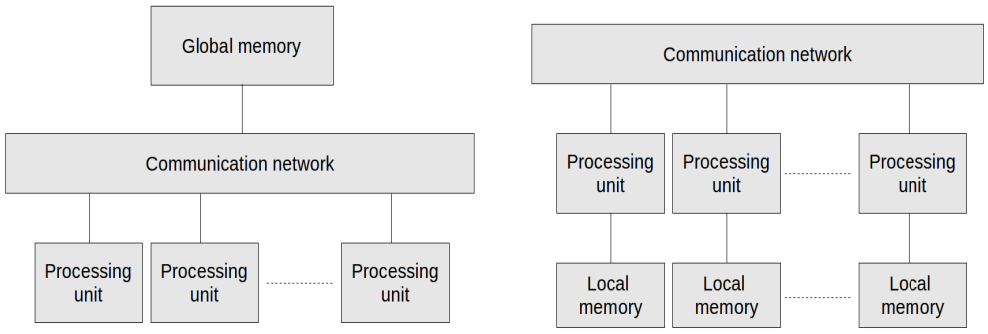


FIGURE 2.2 – Représentation d'une architecture a mémoire partagée (à gauche) et une architecture a mémoire distribuée (à droite)

Mémoire partagée Dans cette architecture représentée à la figure 2.2, tous les unités de traitement utilisent la même mémoire, c'est-à-dire qu'elles partagent la même mémoire d'où le nom de mémoire partagée (Shared Memory) [33]. L'accès à cette mémoire peut être réalisé par différents moyens (bus, crossbar,...). Ce type d'architecture mémoire présente certains avantages dont la communication entre processeurs étant donné qu'ils partagent la même mémoire. L'inconvénient de cette approche est la bande passante de la mémoire : en effet, dans une architecture à grand nombre de processeurs, l'accès à la mémoire devient très lent, ce qui réduit considérablement la performance des processeurs et donc du système total. Dès lors qu'on atteint une dizaine de processeurs dans le réseau, la connexion entre unité de traitement et la mémoire constitue alors le goulot d'étranglement.

Mémoire distribuée Dans cette architecture représentée à la figure 2.2, la mémoire est distribuée sur toutes les unités de traitement, c'est-à-dire qu'elles ont chacune leur propre mémoire, d'où le nom de mémoire distribuée (Distributed Memory) [33]. L'avantage de cette approche est la bande passante de la mémoire qui reste totale pour chaque processeur. Cependant, le point négatif de cette approche est la communication entre processeurs qui devient non-triviale et peut constituer le goulot d'étranglement du système. Avec cette architecture mémoire, les processeurs sont généralement reliés par un

réseau de communication avancé généralement de type NoC (Network-on-Chip) afin de répondre au problème de communication entre processeurs. Ce sujet a fait l'objet de nombreux travaux qui permettent de répondre de façon satisfaisante à ce problème ce qui explique que cette architecture mémoire est aujourd'hui la plus utilisée dans les systèmes multiprocesseurs à grande dimension.

2.5 Homogénéité vs. Hétérogénéité

Une architecture homogène est composée de nœuds (élément de base de calcul du réseau multiprocesseurs) identiques contrairement à une architecture hétérogène composés de nœuds différents. L'avantage d'une architecture hétérogène réside dans la possibilité de la rendre spécifique aux besoins d'une application donnée permettant ainsi d'atteindre des performances optimales. A contrario, cette spécificité à une application (ou un type d'application) la rend moins efficace pour d'autres types applications, ce qui réduit l'adaptabilité et la flexibilité de ces d'architectures. Les architectures hétérogènes souffrent également de la complexité de la programmation, de la difficulté à gérer les différents domaines d'horloge (entre les différents composants du système) et de la diversité des interfaces d'entrés/sorties qui complique les communications entre les composants. Ce dernier point est très pénalisant dans un domaine où la notion de délai de mise sur le marché est cruciale car il tend à compliquer et du coup à rallonger la phase de conception. Toutefois, la technique de réutilisation ("reuse") de composants (blocs IP) lors de la conception des architectures hétérogènes peut atténuer cet aspect. Ces inconvénients sont inexistants ou nettement plus faibles dans une approche homogène. La figure 2.3 montre le degré d'homogénéité des systèmes et leur granularité. Ce dernier terme fait intervenir la notion de grain faisant référence à la taille du composant : un composant plus petit (grain fin) est un composant dédié et spécifique à une tâche particulière, cette spécificité rend le composant plus performant pour accomplir la tâche, contrairement à un élément plus généraliste (grain fort).

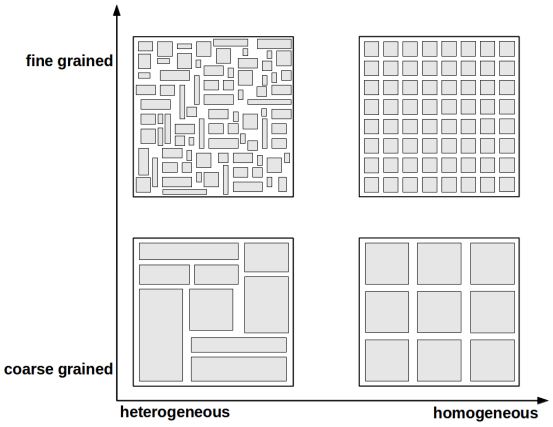


FIGURE 2.3 – Représentation de la granularité en fonction de l'homogénéité

2.6 État de l'art sur les MPSoC

Dans cette section, les projets les plus cités dans le domaine des multiprocesseurs sur cible ASIC sont abordés. Deux projets récents sont plus particulièrement étudiés : le premier est développé par Kalray [34] et le deuxième par Adapteva [35]. Les travaux réalisés par ces deux startup innovantes bien que rarement cités par des travaux de thèse sont récents et sont très connexes au travail et aux objectifs du présent travail.

2.6.1 RAW (MIT en 1997)

Le MIT (Massachusetts Institute of Technology) propose une architecture multiprocesseurs appelé RAW [36]. Le circuit est constitué de 16 tuiles (brique de base du réseau multiprocesseurs) en topologie grille. Chaque tuile est constituée d'un processeur, d'un routeur avec deux types de communication (statique et dynamique), d'un réseau d'interconnexion, des mémoires distribuées (deux mémoires pour les instructions (32Ko et 64Ko) et une mémoire pour les données (32Ko)). Le circuit possède également un accès à des mémoires partagées externes de type DRAM via 8 ports distincts.
Fabriqué dans une technologie CMOS 0.18μm de chez IBM, la surface totale occupée est de 16mm x 16mm. La consommation moyenne du circuit est de 18.2W à une fréquence de 425MHz. Le boîtier utilisé est de type GCAC (Ceramic Column Grid Array)) à 1657 broches, avec 1080 broches d'entrées/sorties de type HSTL (High Speed Transceiver Logic). La figure 2.4 présente le layout du circuit.

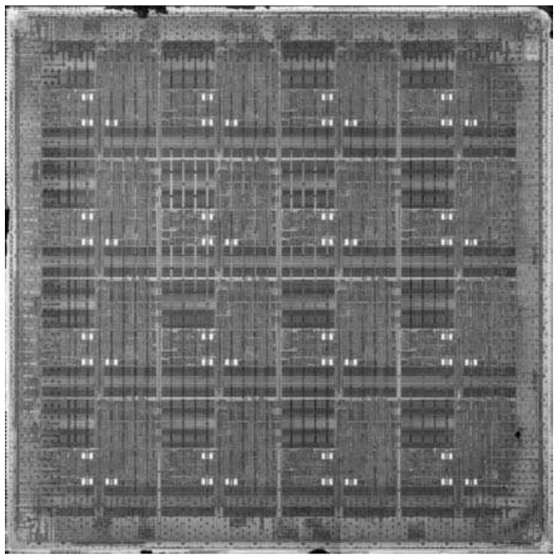


FIGURE 2.4 – Layout du RAW [36]

2.6.2 Tile64 (Par Tilera en 2006)

La société Tilera propose un circuit multiprocesseurs appelé Tile64 [37]. Il est basé sur les travaux du projet RAW du MIT. Le circuit cible la haute performance exigé un large éventail d'applications embarquées tel que des applications multimédia numériques et réseaux. Le circuit est constitué de 64 processeurs en topologie grille (8 X 8). Le processeur est à architecture 32 bits VLIW (Very Long Instruction Word) avec une mémoire distribuée. Les processeurs sont connectés entre eux à l'aide d'un routeur comme présenté dans la figure 2.5. Le routeur offre cinq types de communications (réseau statique, réseau dynamique d'utilisateur, réseau dynamique de tuile, réseau dynamique de mémoire et réseau dynamique d'entrées/sorties) dont chacune est bidirectionnelle et de 32 bits. Le circuit possède un accès à des mémoires partagées externes grâce à 4 contrôleurs DDR implantés dans le circuit. La bande passante maximale pour accéder à ces mémoires est plus de $25GB/s$ (en utilisant les quatre interfaces 72 bits à une fréquence de 800MHz). Le circuit peut fonctionner à une fréquence comprise entre 500MHz et 866MHz. La puissance consommée à une fréquence de 700MHz est comprise entre $15W$ et $22W$. La fabrication de ce circuit a été réalisée en technologie CMOS $90nm$. La figure 2.5 présente l'architecture du circuit.

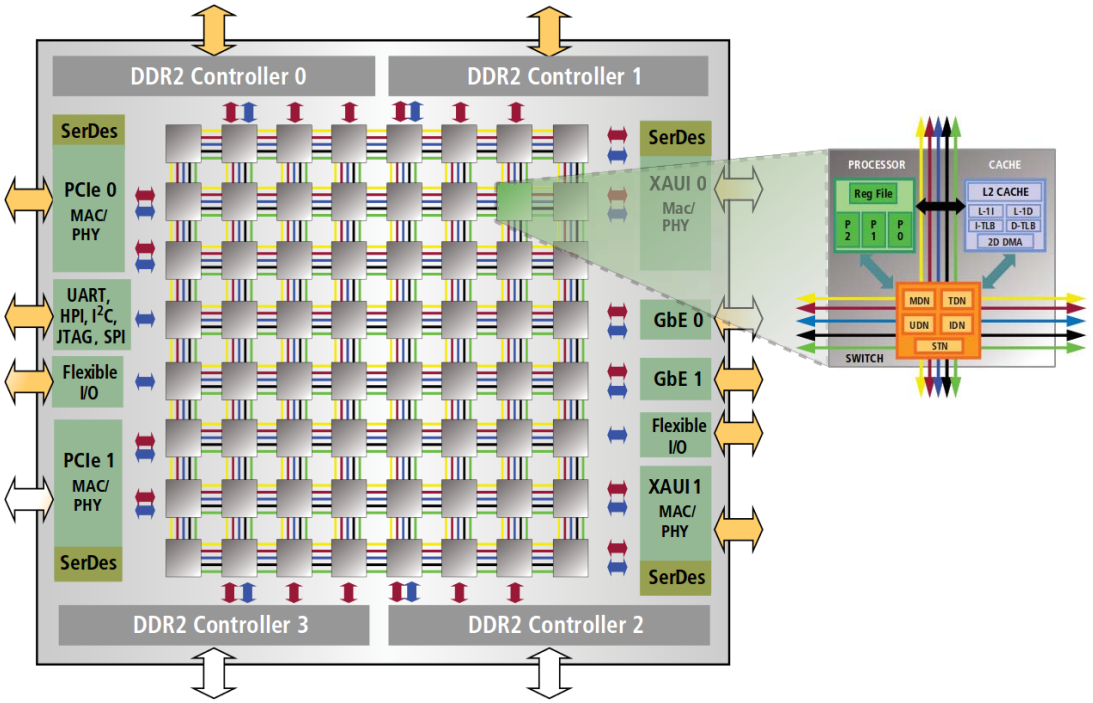
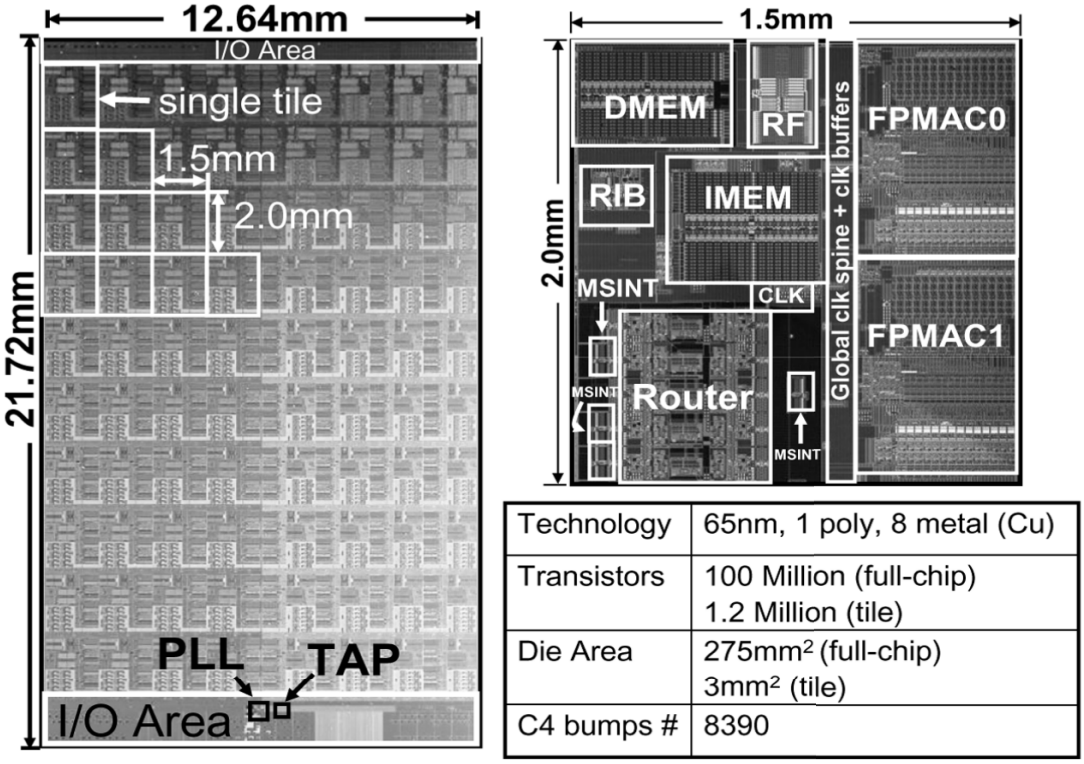


FIGURE 2.5 – Architecture du Tile64 [37]

2.6.3 Teraflops (Intel en 2007)

Le TeraFLOPS [38] est un circuit à 80 nœuds connectés en topologie grille (8 x 10). Les nœuds sont constitués d'un cœur avec une unité à virgule flottante et des routeurs de commutation, les deux étaient conçus pour fonctionner à une fréquence de $4GHz$. Les nœuds sont à mémoires distribuées (instructions et données). Chaque nœud a deux unités Multiplicateur-Accumulateur à virgule flottante pipelinée et à simple précision (FPMAC pour Floating-Point Multiply ACcumulators), qui disposent d'une boucle d'accumulation à cycle unique pour un haut débit. Le réseau en grille offre une bande passante de $2Terabits/s$. La fabrication du circuit a été réalisée dans une technologie CMOS de $65nm$. La surface de la puce conçue est de $275mm^2$, elle contient 100 millions de transistors. Le TeraFLOPS est considéré comme le premier circuit sur silicium entièrement fonctionnel qui atteint plus de $1Tflops$ de performances. La consommation du circuit est de $97W$ à une fréquence de 4.27 GHz et sous une tension d'alimentation de $1.07V$. La figure 2.6 présente le layout du circuit.



Technology	65nm, 1 poly, 8 metal (Cu)
Transistors	100 Million (full-chip) 1.2 Million (tile)
Die Area	275mm² (full-chip) 3mm² (tile)
C4 bumps #	8390

FIGURE 2.6 – Layout du Teraflops [38]

2.6.4 AsAP 1 (Univ. Californie en 2008)

L'Asynchronous Array of simple Processors dit AsAP [39] de l'Université de Californie est un circuit intégré multiprocesseurs basé sur une topologie grille de 36 processeurs asynchrones (6 x 6) avec mémoires distribuées (instructions et données). L'architecture du processeur est constituée de 9 étages de pipeline, programmable et flexible. Les communications entre processeurs concernent les plus proches voisins. Chaque processeur dispose de deux ports d'entrées asynchrones et peut connecter chaque port à l'un de ses quatre processeurs voisins les plus proches. Les connections d'entrée de chaque processeur sont définies au cours de la séquence de configuration après la mise sous tension. Pour ce qui est de la sortie, chaque processeur dispose d'un port de sortie qui peut être modifié en importe quelle combinaison des quatre processeurs voisins à tout moment par le logiciel.
Chaque processeur est cadencé indépendamment par des oscillateurs dans une méthodologie GALS (Globally Asynchronous Locally Synchronous) [1]. Les GALS qui gèrent l'horloge et la communication entre le plus proche/voisin améliore fortement l'extensibilité de l'architecture et offrent des possibilités d'atténuer les effets des variations/-modifications de l'architecture, les limites liées au problème d'interconnexion et les échecs du processeur. La consommation moyenne d'un seul processeur à une fréquence de 116MHz et une tension d'alimentation de $0.9V$ est de $2.4mW$. Si nous passons à une fréquence et une tension plus élevées (475MHz et $1.8V$) la consommation d'un seul processeur passe à $32mW$. Le circuit a été fabriqué avec une technologie CMOS 0.18μm de chez TSMC. Chaque processeur occupe une surface de $0.66mm^2$ et peut fonctionner à une fréquence comprise entre $520-540$MHz avec une tension d'alimentation de $1.8V$. La figure 2.7 montre le layout du circuit ASAP avec ses principales caractéristiques.

2.6.5 AsAP 2 (Univ. Californie en 2009)

Une seconde génération de AsAP [40] contenant 167 processeurs a été fabriquée en technologie CMOS $65nm$ de ST-Microelectronics. Parmi les 167 processeurs, il y a 164 processeurs programmables occupant une surface de $0.17mm^2$. Le circuit contient trois blocs "mémoire partagée" de 16 Ko chacune. Les processeurs peuvent fonctionner à une fréquence maximale de $1.2GHz$ sous une tension d'alimentation de $1.3V$.
A $1.2V$, les processeurs fonctionnent à une fréquence de $1.07GHz$ et consomment $47.5mW$. La figure 2.8 montre le layout du AsAP avec ces principaux caractéristiques.

1. Une méthodologie d'horloge synchrone à l'échelle globale (circuit globalement synchrone) est souvent utilisée dans les circuits intégrés modernes. Mais avec les retards sur les fils d'interconnexions et les variations des paramètres technologiques (surtout avec les nœuds technologiques fortement submicroniques), il est devenu de plus en plus difficile de concevoir des puces de grande taille fonctionnant à une fréquence d'horloge élevée (problème explosion de la consommation dynamique). En outre, le style synchrone n'a pas la souplesse de contrôler indépendamment la fréquence d'horloge entre les sous-composants du système pour atteindre une efficacité énergétique. Au contraire, le style d'horloge totalement asynchrone a comme caractéristique principale d'améliorer la vitesse et la puissance, mais il lui manque actuellement le support d'une chaîne d'outil CAO ce qui rend difficile la conception et, par voie de conséquence, réduit son efficacité.

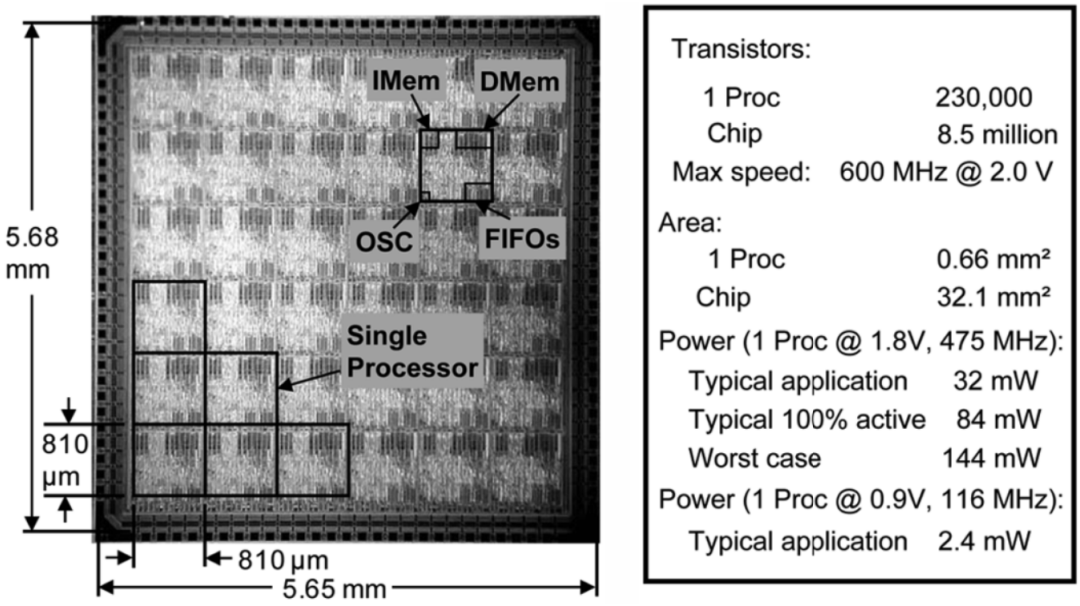
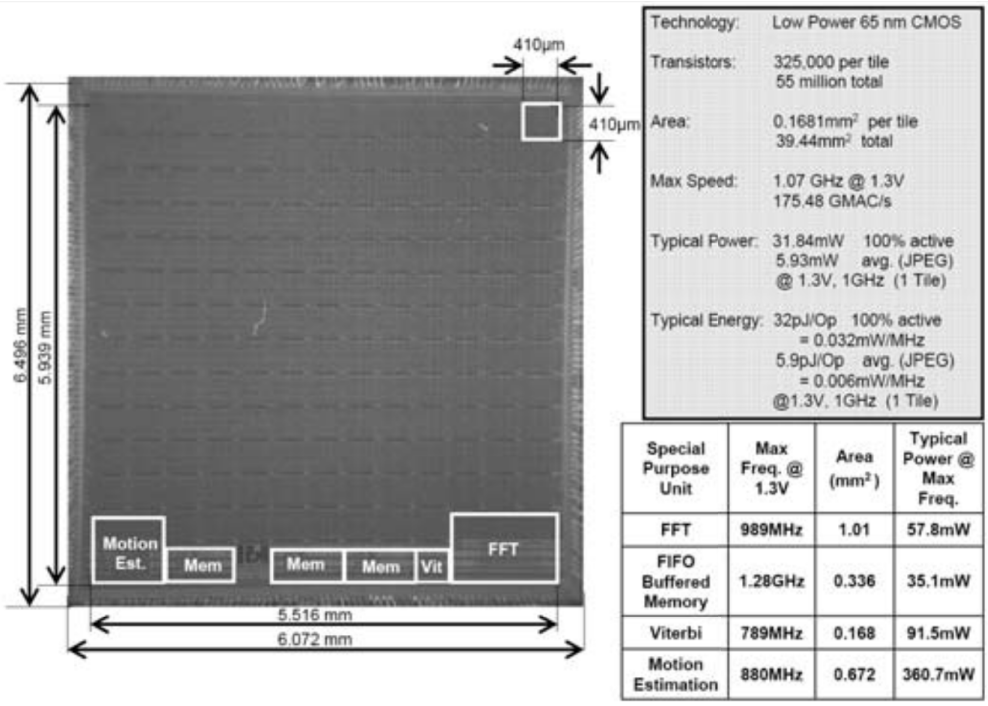


FIGURE 2.7 – Layout du ASAP1 avec les principaux caractéristiques [39]



Special Purpose Unit	Max Freq. @ 1.3V	Area (mm²)	Typical Power @ Max Freq.
FFT	989MHz	1.01	57.8mW
FIFO Buffered Memory	1.28GHz	0.336	35.1mW
Viterbi	789MHz	0.168	91.5mW
Motion Estimation	880MHz	0.672	360.7mW

FIGURE 2.8 – Layout du circuit AsAP2 et ses principales caractéristiques [41]

2.6.6 EpiphanyIII (Adapteva en 2011)

La startup Adapteva [35] propose un circuit multiprocesseurs de 16 cœurs en technologie $65nm$. L'architecture est à mémoire distribuée, partagée et extensible [35]. Les nœuds sont connectés par un réseau de topologie grille. La figure 2.9 montre l'architecture du EpiphanyIII.

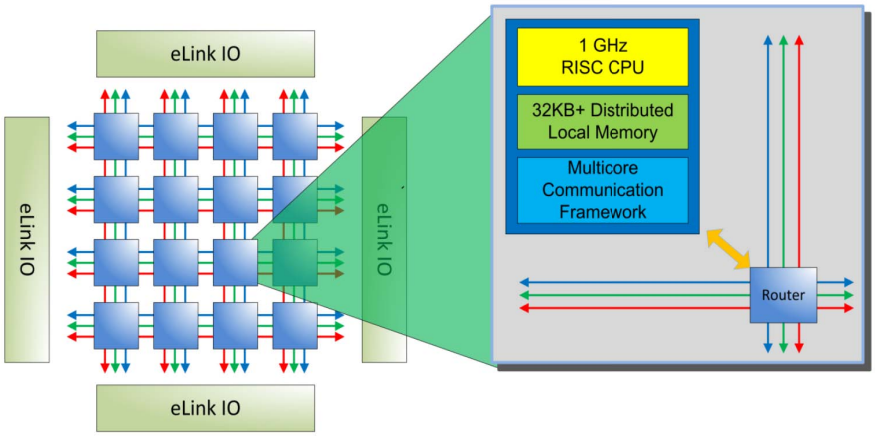


FIGURE 2.9 – Architecture du Epiphany-III [35]

Processeur et système de mémoire Le circuit comprend 16 cœurs composés d'un CPU RISC avec unité de virgule flottante. Le cœur nommé eCore est capable de deux opérations en virgule flottante par cycle d'horloge et un calcul entier par cycle d'horloge. La CPU a un jeu d'instructions généraliste efficace dans les applications gourmandes en calcul, tout en étant efficacement programmable en C/C++.
L'architecture mémoire Epiphanie est basée sur un plan de mémoire partagée dans lequel chaque processeur peut accéder à sa propre mémoire locale et à la mémoire d'autres processeurs par des instructions régulières de chargement/stockage (load/store). Le système de mémoire locale est composé de 4 sous-banques distinctes ce qui permet un accès simultané à la mémoire grâce aux étapes de chargement d'instruction(fetch), de chargement/stockage (load/store) et par des transactions de mémoire entrepris par le DMA des autres processeurs dans le système. La figure 2.10 montre un schéma bloc du eCore, le nœud de base du circuit EpiphanyIII.

Réseau sur puce Le réseau sur puce (NoC pour Network on Chip) du circuit est appelé eMesh (fig. 2.11). C'est un réseau à topologie grille qui gère l'ensemble des communications externes et internes au circuit. Le réseau eMesh utilise une transaction de mémoire 32 bits et se compose de trois structures de mailles distinctes et orthogonales, chacune desservant différents types de trafic : un réseau pour le trafic d'écriture dans la puce, un réseau de trafic d'écriture externe au circuit et un réseau pour tous les trafics de lecture.

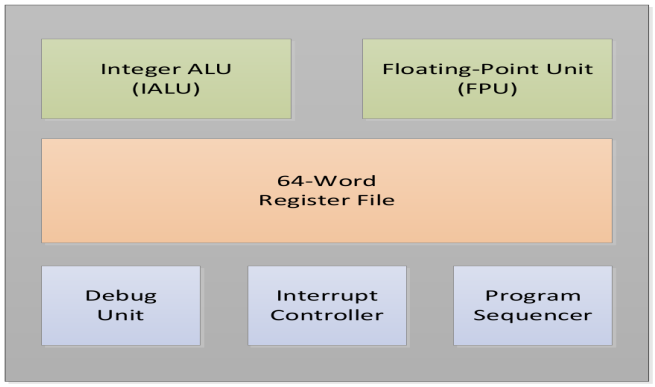


FIGURE 2.10 – Schéma bloc de l'architecture du ecore [35]

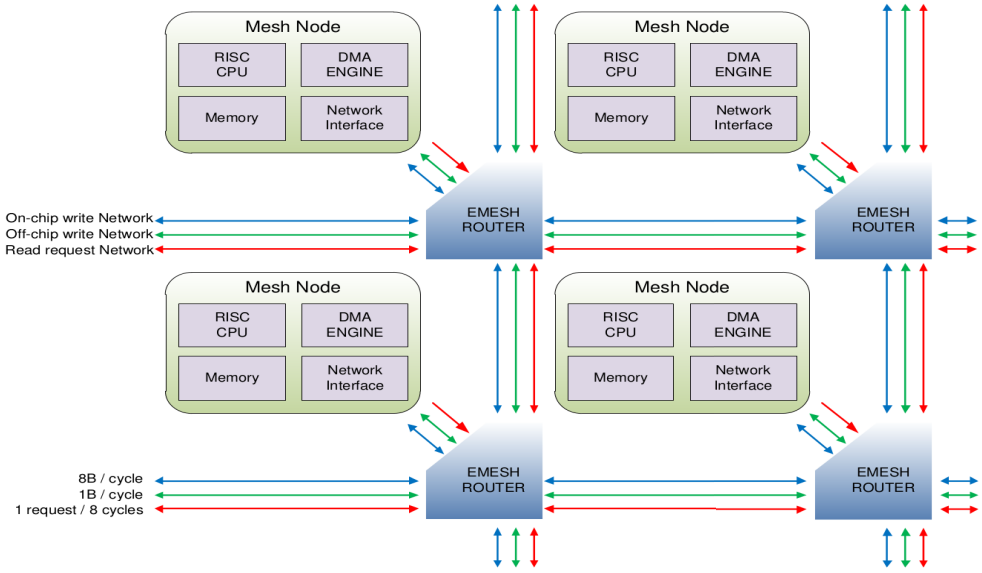


FIGURE 2.11 – Architecture du emesh [35]

Résultats d'implantation EpiphanyIII est fabriqué en technologie $65nm$. La surface est de $8,96mm^2$ et les cœurs fonctionnent à une fréquence de $1GHz$. La consommation maximale est estimée à $2W$. Le pic de performance est de $32GFLOPS$. Les bandes passantes du circuit sont : $512GB/s$ pour la mémoire locale, $64GB/s$ pour le réseau sur puce (NoC), $8GB/s$ pour le hors puce (Off-Chip). Le circuit contient $0,5MB$ de mémoire totale pour le circuit (mémoires partagée et distribuée). Le boîtier est un BGA 324-ball (15x15mm). La figure 2.12 montre le circuit EpiphanyIII référence E16G301 :

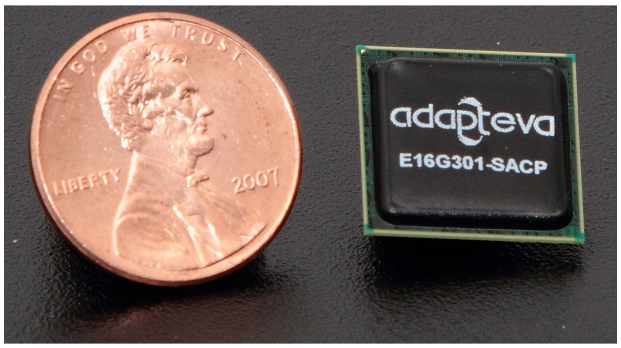


FIGURE 2.12 – Epiphany-III 16-core 65nm Microprocessor (E16G301) [35]

2.6.7 MPPA 256 (Kalray en 2012)

Le MPPA256 (Multi-Purpose Processor Array) est le premier circuit de la famille des multi-cœurs proposés par Kalray [34]. Le circuit est constitué de 256 MPPA cœur. Les cœurs sont regroupés dans 16 clusters (à raison de 16 cœurs par cluster). Le circuit contient aussi 4 sous-systèmes pour la gestion des entrées/sorties, eux-mêmes reliés entre eux par deux NoC. La figure 2.13 présente une vue d'ensemble de l'architecture du MPPA256. Afin d'expliquer l'architecture avec plus de détails, nous illustrons l'architecture en trois niveaux : tuile, cluster et circuit.

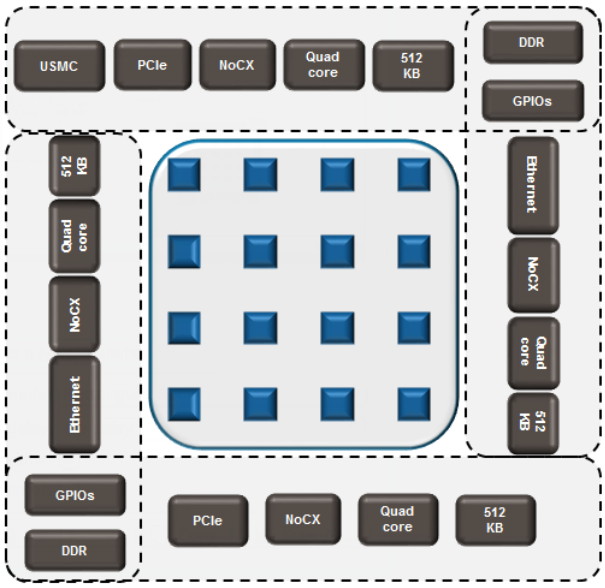


FIGURE 2.13 – Vue d'ensemble de l'architecture du MPPA256 [34]

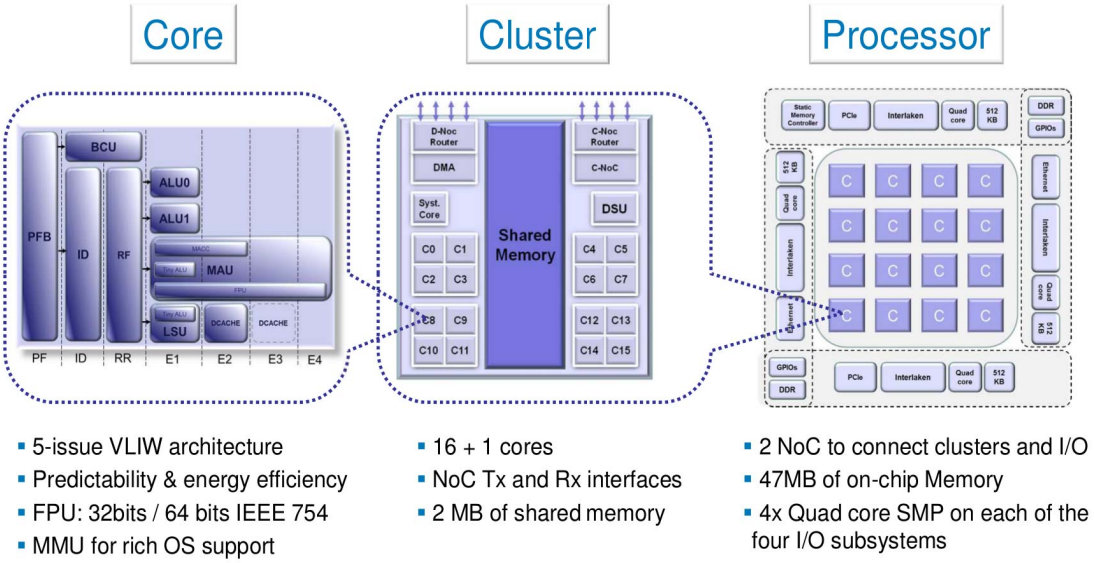


FIGURE 2.14 – Architecture du MPPA256 [34]

Niveau tuile Le processeur utilisé dans le circuit MPPA256 est basé sur une architecture 32 bits de la famille VLIW. Chacun de ces processeurs élémentaires contient une unité de branchement/control (Branch/Control), deux unités arithmétiques logiques, une unité de chargement/sauvegarde (Load/Store) avec une ALU simplifiée, une unité Multiplicateur-Accumulateur (MAC)/ FPU (Float Point Unit) avec une ALU simplifiée, une FPU standard de la norme IEEE 754-2008 avec une multiplication-addition fusionnée ((FMA) Fused Multiply-Add) et une unité de gestion de mémoire MMU (Memory Management Unit). Cette architecture permet d'exécuter jusqu'à cinq opérations entières de 32 bits par cycle d'horloge.

Niveau cluster Chaque cluster de calcul est composé de 16 cœurs (PE cores) identiques avec une FPU et une MMU. La tension est ajustée dynamiquement (Dynamic Voltage) et des techniques d'ajustement de fréquence (Frequency Scaling) et de mise en veille dynamique de la tension d'alimentation (Dynamic Power Switch off) sont mises en œuvre dans chaque cluster. Le cluster contient un $17^{ième}$ cœur (RM core) avec une FPU et une MMU privées (ce cœur est appelé cœur système). Pour ce qui est de la mémoire, un cache d'instruction et de données de type L1 pour chaque cœur, une unité d'accès direct à la mémoire (Direct Memory Access (DMA)) et une mémoire partagée de $2Mo$ sont mis en œuvre. Les cœurs sont reliés soit à une mémoire multi-banques permettant une faible latence lors de l'accès ou soit à une banque à accès privé en fonction de la configuration. Le cluster contient aussi une unité de débogage.

Niveau circuit Le NoC a globalement une structure de tore 2D fournissant une bande passante bidirectionnelle jusqu'à $3,2GB/s$ entre chaque cluster adjacent. Le

NoC garantit des temps de latence déterministes pour tous les transferts de données.

Interfaces et Performances Le circuit communique avec des dispositifs externes par le biais de sous-systèmes d'entrées/sorties situés à la périphérie du NoC. Les sous-systèmes d'entrées/sorties sont mis en œuvre via diverses interfaces standard. Nous pouvons citer :

- Deux interfaces DDR3 (chaque canal est de $64bits$ avec un code correcteur ECC en option et fournit jusqu'à $12,8Go/s$).
- Deux PCIe Gen3 X8 intégrant chacun un DMA avancé avec un support (scatter/gather) offrant un transfert de données efficace.
- Deux contrôleurs Ethernet intelligents qui peuvent être configurés pour fournir des interfaces 4x1GbE, 4x10GbE ou 1x40GbE.
- Un contrôleur de mémoire statique universel qui permet de connecter jusqu'à cinq périphériques externes comme des mémoire de type : NAND/NOR Flash, Flash série et SRAM asynchrones.
- Deux banques de 64 entrées/sorties à usage général (General Purpose I/O), chaque banque peut être configurée en PWM, UARTs, SPI ou I2C. Ces banques peuvent également fonctionner en mode d'accès direct au réseau (Direct Network Access), fournissant une interface à très faible latence.
- Une interface NoC Express (nommée NoCX) fournit une bande passante totale de $40Gb/s$. La NoCX offre la possibilité de faire évoluer facilement le nombre de cœurs par la connexion de plusieurs processeurs MPPA. La NoCX est également un moyen efficace pour coupler la MPPA avec un FPGA externe utilisé en tant que co-processeur ou d'un pont d'interface.

Le circuit est présenté comme pouvant atteindre environ $200GOPS$ à une fréquence d'horloge de 400MHz avec une consommation d'énergie d'environ $5W$ [20] ($10W$ au maximum). Le circuit est fabriqué avec une technologie CMOS $28nm$ de chez TSMC [42]. Avec environ 3 milliards de transistors, la surface de la puce est d'environ $300mm^2$ [43]. La figure 2.15 montre le circuit MPPA 256 :

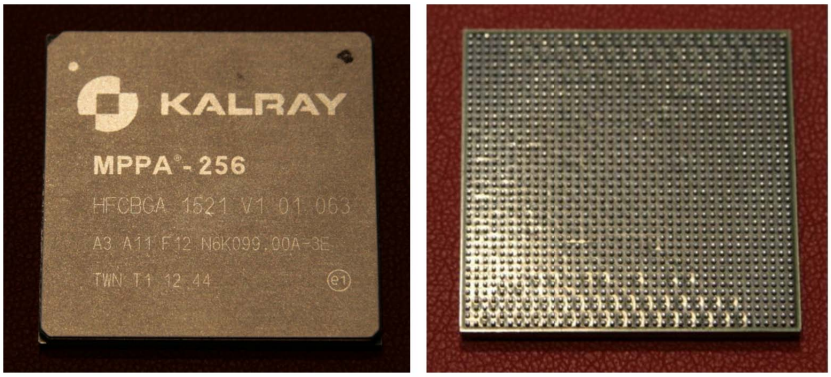


FIGURE 2.15 – MPPA intégré dans un boitillé a 1600 broches [43]

2.6.8 EpiphanyIV (Adapteva en 2014)

Le deuxième circuit est appelé EpiphanyIV. Le circuit est fabriqué en technologie $28nm$ et contient 64 cœurs (fig. 2.16). La surface totale est de $8,2mm^2$. EpiphanyIV est une extension du EpiphanyIII avec un nombre de cœurs quatre fois plus grand pour une même consommation. Ce gain en terme de performance est expliqué par la migration de la technologie $65nm$ à la technologie $28nm$. Les cœurs fonctionnent à une fréquence de 800MHz. Le pic de performance est de $102GFLOPS$. Les bandes passantes du circuit sont : $1,6TB/s$ pour la mémoire local, $102GB/s$ pour le réseau sur puce (NoC) et $6,4GB/s$ pour le hors puce (Off-Chip). Le circuit contient $2MB$ de mémoire sur puce partagée distribuée.

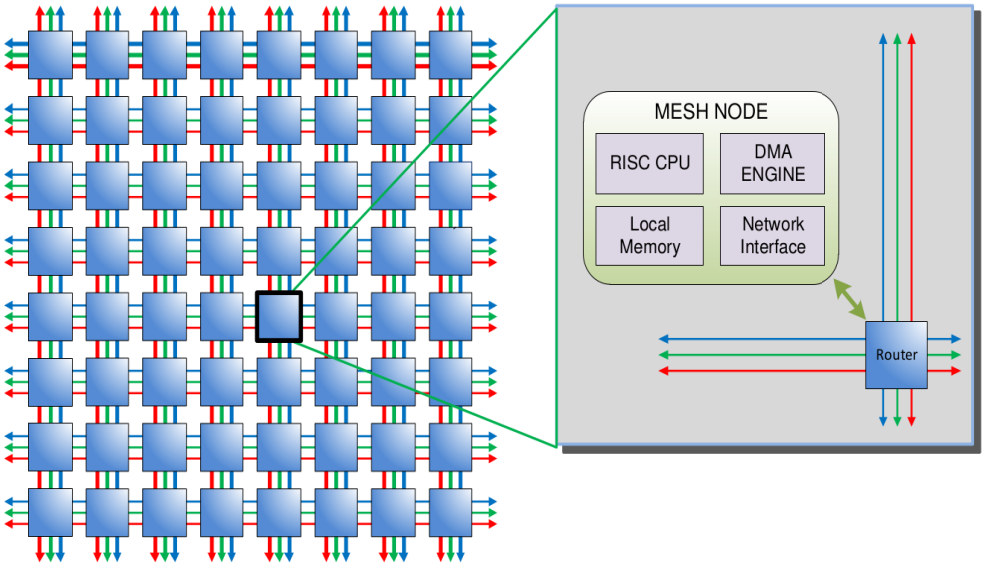


FIGURE 2.16 – Architecture du Epiphany-IV [35]

Le boîtier utilisé est le même que pour le circuit EpiphanyIII. La figure 2.17 montre le Epiphany-IV 64-core 28nm Microprocessor (E64G401) :

2.6.9 MPPA-512 et MPPA-1024 (Kalray en 2015)

Les MPPA 512 et MPPA 1024 sont prévus fin 2015 grâce au recours au nœud technologique $16nm$. A travers cette migration, Kalray revoit la conception afin de doubler la puissance en FPU double précision (voir la tripler en simple précision) mais également pour viser une fréquence de $700-800$MHz [43]. Les puces multicoeurs MPPA possèdent jusqu'à un millier de processeurs sur une seule puce. Ils sont disponibles dans différentes configurations afin de répondre aux exigences des clients en terme d'applications (tab. 2.2). Tous les produits de la série multicoeurs MPPA disposent de techniques d'ajustement dynamique de la tension et d'ajustement de fréquence ce qui permet une dissipation de puissance et des performances de calcul optimums [34].

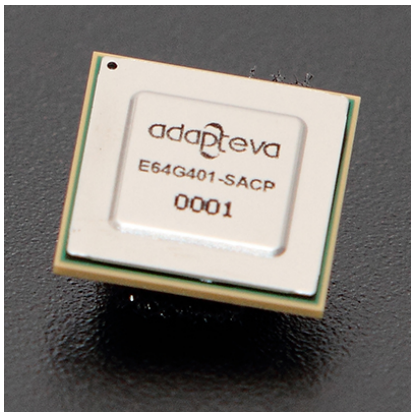


FIGURE 2.17 – Epiphany-IV 64-core 28nm Microprocessor (E64G401) [35]

Génération du cœur	Nombre de cœurs de traitement	GFLOPS/W	GOPS/W
Andey	256	25	75
Bostan (2014)	256	50	80
Coolidge (2015)	64/256/1024	75	115

TABLE 2.2 – Feuille de route de la série MPPA multicoeurs [34]

2.7 Situation du projet HNCP

C'est dans ce contexte que nous allons présenter un circuit multiprocesseurs sur une cible ASIC avec une technologie fortement intégrée au même niveau que les projets présentés dans la section précédente. Les deux derniers projets présentés sont menés par deux statup crées récemment, ce qui montre que le domaine est en forte expansion et en recherche de nouvelle idées et de solutions innovantes pour répondre aux besoins croissants en terme de puissance de calcul. Un aspect non négligeable de ce type de projet réside dans le nombre de personnes impliqué sur le projet et le budget déployé. Par exemple, une startup comme Kalray emploie actuellement 55 ingénieurs. Fin 2010, pour financer son programme de recherche et développement cette startup a levé 6 millions d'euros, après avoir mobilisé, en 2008, un premier tour de table de 2,8 millions d'euros, complété par un financement ISI d'Oséo [42].

Pour ce qui est de la conception du circuit, notre projet se démarque par 2 caractéristiques fortes. La première est que notre circuit est dédié à un domaine bien précis qui est le traitement d'image ce qui oriente la définition des choix architecturaux afin de répondre au besoin du domaine et ainsi atteindre une performance maximale. La deuxième caractéristique est que la solution proposée repose sur une architecture logicielle dédiée contrairement à tous les projets présentés précédemment qui utilisent des systèmes d'exploitation embarqués tels que linux. L'approche proposée est notamment basée sur le concept des squelettes de parallélisation présenté dans le chapitre 1.

Notre laboratoire n'ayant, avant ces travaux, aucune expérience dans la fabrication de circuit ASIC, cette thèse constitue les premiers travaux locaux dans ce domaine. L'idée forte des travaux présentés dans cette thèse est l'étude de la faisabilité de la méthodologie HNCP sur ASIC.

2.8 Conclusion

Dans ce chapitre, nous avons présenté quelques notions de base liées au domaine des MPSoC. Un état de l'art sur les projets multiprocesseurs sur cible ASIC les plus significatifs de ces dernières années est présenté. Dans cet état de l'art, nous avons présenté les performances qu'on peut atteindre à ce jour sur une cible ASIC. Le gain en performance atteignable en augmentant le nombre de cœurs et en migrant d'un nœud technologique à un nœud technologique plus intégré est également illustré (par exemple le passage de EpiphanyIII vers EpiphanyIV).
Nous avons la confirmation via notamment les circuits réalisées par les deux startup (Kalray et Adapteva) que le chemin entrepris par notre équipe depuis une dizaine d'année (c'est-à-dire l'approche multiprocesseurs homogènes communicants HNCP) est réalisable sur des technologies fortement intégrées et que nous pouvons atteindre un nombre de cœurs dépassant le millier sur une même puce. Cet objectif ambitieux doit évidemment aboutir à plus de performance et plus de simplicité lors de la mise en œuvre que les travaux précédents (basés sur des cibles FPGA) notamment pour ce qui est de l'aspect logiciel (parallélisation d'application).

Chapitre 3

Passage FPGA-ASIC : création du nœud de base du MPSoC

3.1 Introduction

Dans le chapitre précédent, nous avons notamment présenté les cibles technologiques FPGA et ASIC. Nous avons présenté la méthodologie de prototypage rapide HNCP qui est basée originellement sur une cible FPGA puis nous avons introduit les limites de cette approche. En se basant sur une comparaison entre les deux cibles technologiques (FPGA vs ASIC) et sur les performances des projets multiprocesseurs présentés dans l'état de l'art, il a été montré que que le passage vers la cible ASIC est la solution technique à suivre afin d'atteindre de meilleurs performances.

Au début de ces travaux de thèse, le flot de conception associée à la méthodologie HNCP, était basé sur des outils et des IPs de chez Xilinx et Altera. Le flot de conception Xilinx est basé sur des outils de conception tel que ISE (Integrated Software Environment) et EDK (Embedded Development Kit). Les IPs utilisés en partie dans ce projet proviennent également de chez Xilinx. Par exemple, le processeur instancié est le MicroBlaze et les mémoires sont des BRAM (soient des IP propriétaire Xilinx). L'utilisation de ces IPs qui a certes facilité dans un premier temps la mise en œuvre de la méthodologie HNCP implique une dépendance avec l'un ou l'autre de ces fournisseurs car ces IPs ne sont pas libres de droit.

Afin de migrer d'une cible FPGA vers une cible ASIC, deux solutions sont possibles : la première est d'acquérir des IPs (par exemple le processeur ARM) et la deuxième est de se tourner vers des IPs libre de droit. La première solution n'a pas été retenue pour deux raisons principalement : premièrement, ces IPs ont un coût dissuasif et deuxièmement ces IPs sont fournies sous forme de boite noir (black box) dont le code source est, pour la plupart, fermé et donc peu évolutif (point bloquant dans la méthodologie HNCP). C'est pour ces raisons que nous avons opté pour la deuxième possibilité basée sur le monde open source qui au-delà du fait que ces projets correspondent à

nos besoins en terme d'investissement financier (gratuit) et de la diversité/richesse des projets, contribue à la recherche et au développement scientifique [44].

Dans ce chapitre, nous définissons la brique de base du futur MPSOC, appelé **nœud**. Cette définition entraîne la description de la démarche réalisée dans le choix des IPs de la communauté libre, leur intégration à la méthodologie HNCP et leur évolution pour les faire correspondre à notre cahier des charges.

Le cahier des charges est le suivant :
- Processeur RISC compatible avec le jeu d'instructions du MicroBlaze. Cette compatibilité permet de conserver la partie logicielle de la méthodologie HNCP. Elle permet également de faciliter la communication entre le processeur et le module DMA-Routeur via des liens point-à-point (FSL).
- Flexibilité dans le choix du mode de communication entre le module DMA-Routeur (mise en œuvre du squelette FARM) et des liens point-à-point (FSL) (mise en œuvre du squelette SCM).
- Possibilité de calcul en virgule flottante.
- Gestion des domaine d'horloge lors des communications intra-nœud et inter-nœud.
- Mémoires double port afin d'avoir un accès direct par le processeur.
- Système de programmation et de debug basé sur le protocole JTAG.
- Système de gestion du flot vidéo.

Dans ce chapitre, nous commençons par présenter le choix du processeur (le cœur du nœud) et sa description architecturale. Ensuite, nous abordons les améliorations effectuées sur ce processeur en terme de communication (ajout d'instruction utilisateur) et en terme de fonctionnalité (ajout de FPU). Puis l'organisation de la mémoire ainsi que sa programmation (par ajout d'un module JTAG) est présentée. Les bus de communication utilisés (FSL et Wishbone) ainsi que la communication par DMA-Routeur est tout particulièrement détaillée. Nous terminons ce chapitre par la description des modules de gestion du flot vidéo.

3.2 Le cœur du processeur

Dans cette section, nous allons dans un premier temps présenter un état de l'art des processeurs disponibles dans la communauté libre puis dans un second temps, proposer une description du processeur choisi.

3.2.1 État de l'art sur les processeurs dans la communauté libre

De nombreux groupes de recherche ont travaillé sur la conception de soft-processeurs libres de droit. Selon le contexte de chaque projet, ces processeurs softcores ont été de plus ou moins maintenus ce qui constitue une première limite pour certains candidats. Un autre problème important est que la plupart de ces softcores sont conçus par des

équipes de recherche indépendantes ce qui fait que les interfaces d'entrées/sorties ne sont pas standardisées. Ceci complique leur utilisation et leur intégration dans un autre projet. La communauté OpenCores regroupe un grand nombre de ces projets de développement de processeur. Un des plus connus est le OpenRISC 1200 [45] qui est basé sur les spécifications de OpenRISC 1000. Il existe tout un ensemble de processeurs 32 bits de type RISC : Plasma [46], aeMB [47], LEON3 [48], OpenFire [49], SecretBlaze [50] et MB-lite [51].

Plusieurs comparaisons ont été faites [52] [53] entre plusieurs processeurs open source sur de nombreux critères (performance, utilisation des ressources, qualité de l'architecture, etc ...). Comparativement au MicroBlaze de Xilinx, les résultats ont montré que le processeurs MB-Lite a des performances très intéressantes (équivalentes entre elles), tout en utilisant moins de ressources. Les résultats ont montré aussi que le SecretBlaze apparaît comme un bon compromis entre la performance et la surface. D'après ces comparaisons, deux processeurs ont été sélectionnés dans le cadre de ce projet pour un complément d'analyse : le MB-Lite et le SecretBlaze.

Afin de répondre aux exigences élevées des MPSoC en terme de communication, les interfaces de communication des deux processeurs ont été évaluées [54]. Les deux processeurs sont compatibles avec le bus Wishbone [55] et intègrent dans leur projet respectif une interface facilitant l'utilisation du bus Wishbone. Pour ce qui est des communication point à point, même si les deux processeurs sont basés sur le jeu d'instructions du MicroBlaze (ou ISA pour Instruction Set Architecture), ils ne disposent pourtant pas de communication point à point. Pour le MB-Lite, l'ajout de FSL était dans la liste des futurs travaux des concepteurs [51]).
Notre choix s'est finalement porté sur le SecretBlaze car il provient de travaux très récents et dispose d'un support réactif du concepteur tout en présentant un compromis performances/surface tout à fait intéressant comparativement à la concurrence.

3.2.2 Description du SecretBlaze

Le SecretBlaze est un processeur softcore 32 bits à jeu d'instructions réduit RISC (Reduced Instruction Set Computer). Ayant une architecture HARVARD, le SecretBlaze dispose de cinq étages de pipeline nommé comme suit : recherche d'instruction, décodage d'instruction, exécution, accès mémoires et retour écriture. La mémoire de données et la mémoire d'instructions sont accessibles via deux bus distincts. Comme nous l'avons déjà évoqué, la communication du SecretBlaze est basée sur le bus de communication Wishbone [55] qui est un bus d'interconnexion libre et largement répandu. Le SecretBlaze ne dispose pas de lien de communication direct par FIFO semblable aux FSL du MicroBlaze [56]. Pour la mise en œuvre d'un réseau multiprocesseurs efficace en terme de performances, il est indispensable de disposer de lien de communication rapide de type FIFO. Nous proposons donc d'adapter le SecretBlaze en ajoutant des instructions utilisateur pour accélérer la communication dans le réseau multiprocesseurs. La figure ci-dessous (fig. 3.1) montre les cinq étages de pipeline du cœur du SecretBlaze.

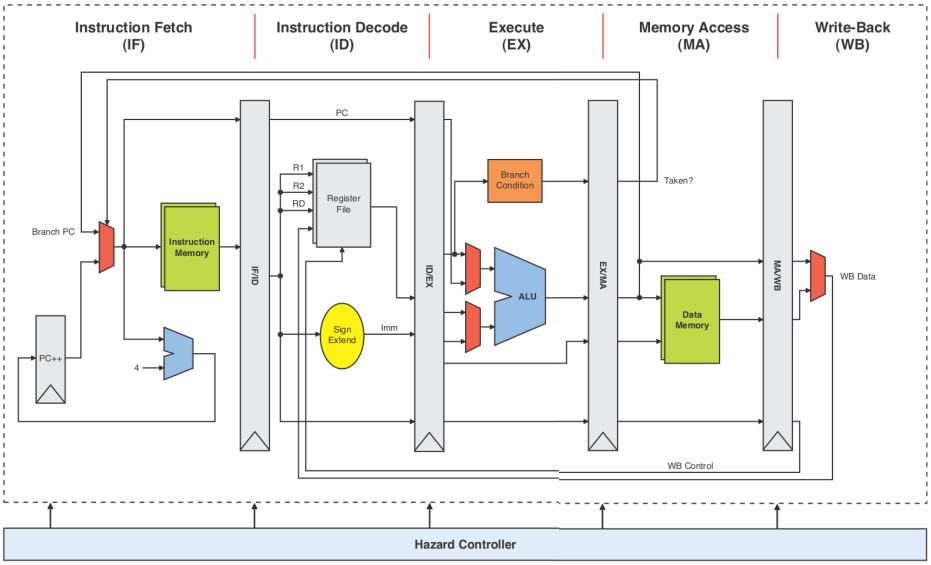


FIGURE 3.1 – Architecture du cœur du SecretBlaze [50]

3.3 Amélioration de la communication par l'ajout d'instructions FSL

Les communications sur les liens FSL [56] se font très simplement grâce à des instructions prédéfinies. Elles sont inclues dans le jeu d'instruction du MicroBlaze. C'est-à-dire que le compilateur mb-gcc (microblaze-gcc), compilateur utilisé pour le MicroBlaze, transforme une instruction FSL en un format que nous pouvons mettre dans le registre d'instruction 32 bits. Globalement, nous pouvons distinguer quatre aspects concernant les instructions FSL :

- Les instructions FSL peuvent faire deux types d'opérations : une écriture (avec les instructions PUT) ou une lecture (avec les instructions GET).
- Les instructions bloquantes ne lisent pas le contenu de la FIFO tant que la FIFO est vide et n'écrivent pas dans la FIFO tant que la FIFO est pleine.
- Deux types de données : donnée simple ou donnée de contrôle (les deux sont de taille 32 bits). Un signal de contrôle est ajouté au protocole de communication afin de les distinguer.
- Le port FSL dans lequel la donnée va être écrite ou lue. Ce dernier est sélectionné de deux façons différentes : fixe dans l'instruction ou dynamique c'est-à-dire que dans l'instruction, il y a la sélection d'un registre qui va contenir le numéro du port.

Ceci constitue seize instructions au total qui sont présentées en détail dans la l'**annexe A.1**. Pour ce faire, deux aspects sont à prendre en compte : l'aspect logiciel et l'aspect matériel.

3.3.1 Résultats d'implantations

Le tableau 3.1 présente l'utilisation des ressources matérielles des implantations entre la version originale et la version améliorée du SecretBlaze sur la cible FPGA Virtex-6 [1]. Comme le montre le tableau, il y a une faible augmentation des ressources matérielles utilisées entre SecretBlaze sans FSL et SecretBlaze avec FSL. Une comparaison avec les ressources matérielles requises pour instancier une FSL du MicroBlaze aurait été intéressante mais, le code source du MicroBlaze n'étant pas disponible, cela fut impossible.

Version du SecretBlaze	SecretBlaze sans FSL	SecretBlaze avec FSL
Nombre de Flip-Flops	901 (0,29%)	913 (0,30%)
Nombre de LUTs	1556 (1,03%)	1677 (1,11%)
Nombre de Slices	516 (1,37%)	580 (1,54%)

TABLE 3.1 – Résultats d'implantations.

3.3.2 Comparaison avec les instructions FSL de MicroBlaze

Pour montrer les performances de la solution proposée (liens FSL sur SecretBlaze), le squelette SCM (voir chapitre 1) est mis en œuvre sur deux architectures. Dans la première architecture proposée sur la figure 3.2 (à gauche), quatre processeurs SecretBlaze sont connectés via des bus FSL. Dans la deuxième architecture proposée sur la figure 3.2 (à droite), quatre processeurs MicroBlaze sont connectés via des bus FSL.

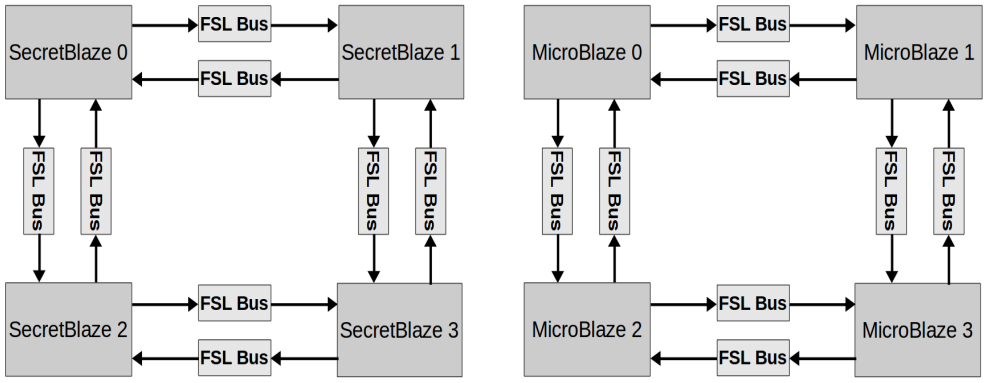


FIGURE 3.2 – Système de communication point à point dans un réseau de quatre processeurs

1. Lorsque le Virtex-6 est cité, cela veut dire : le Virtex-6 XC6VLX240T-FF1156 (speedgrade -1) de technologie 40 nm

Les deux architectures ont été implantées sur une même cible FPGA (Virtex-6). Les deux architectures ont également été synthétisés et placé-routés avec les mêmes outils XST (Xilinx Synthesis Technology) et avec la même fréquence de fonctionnement (200MHz). Le code source C de l'algorithme de seuillage adaptatif est le même dans les deux architectures et les outils de compilation sont réglés avec le même niveau d'optimisation. Le tableau suivant (tab. 3.2) indique les temps de traitement (en microsecondes) des architectures présentées ci-dessus :

Processus SCM	Quatre SecretBlaze connectées via FSL	Quatre MicroBlaze connectées via FSL
Temps de Split (μs)	0,39	0,76
Temps de Compute (μs)	0,24	0,20
Temps de Merge (μs)	1,54	3,16
Temps total (μs)	2,17	4,13

TABLE 3.2 – Comparaison du temps d'exécution entre les deux architectures

Sur le tableau ci-dessus, nous pouvons clairement remarquer que le FSL du SecretBlaze est plus rapide que celui de MicroBlaze (presque 2 fois plus rapide). Le code source de MicroBlaze n'étant pas disponible, il n'est donc pas facile de donner une juste explication de ce résultat. En tout état de cause, la solution proposée se montre très efficace.

3.4 Amélioration de la performance par l'ajout d'unité à virgule flottante

L'utilisation d'une unité à virgule flottante (en anglais Floating Point Unit, soit FPU) est devenue obligatoire dans de nombreuses applications de traitement de données vu que l'étage d'exécution dans les processeurs pipeliné a tendance à être le goulot d'étranglement du système. La possibilité d'ajouter une FPU à un processeur est aujourd'hui une option indispensable, un simple cœur RISC n'étant souvent pas suffisant pour répondre de façon satisfaisante aux attentes de performances élevées dans le domaine des applications modernes et complexes. Considérant ceci, la FPU ajoutée se doit d'être rapide et efficace pour répondre aux contraintes de conception. La méthode proposée a l'intention de faciliter l'ajout d'une FPU s'adaptant au softcore SecretBlaze grâce à l'utilisation d'instructions "utilisateur" qui sont les instructions FSL définie précédemment.
Dans la version native du SecretBlaze, il n'y a pas de FPU disponible. Nous proposons de renforcer le SecretBlaze avec une FPU disponible dans la communauté libre (afin de maintenir le projet libre de droit). De plus, nous proposons une amélioration de l'architecture par l'ajout d'une unité de commande de la FPU. Ce module permet de gérer l'unité FPU dans le but de réaliser des fonctions complexes (trigonométriques, logarithmiques, polynomiale ...).

L'architecture proposée est différente des architectures de type processeur/ co-processeur. Les approches processeur/co-processeur ont de nombreuses limites telles que les problèmes de gestion de communication de données entre le processeur et le co-processeur et la non évolutivité du système (tel est le cas de [26]). Ces limites sont surmontées dans l'architecture proposée grâce à l'utilisation de l'unité de commande de la FPU et des instructions utilisateur de type point à point (FSL). Ce système FPU est suffisamment modulaire pour être facilement personnalisé par l'utilisateur pour une extension (ajout de nouvelles fonctions de calculs).
Dans cette section, nous allons dans un premier temps faire un état de l'art sur les FPU disponibles dans la communauté open source avant d'introduire et de discuter notre choix de FPU. Ensuite, nous présentons la solution implantée, c'est-à-dire la FPU utilisant les instructions FSL. Nous présentons à la fois les optimisations de l'architecture proposée mais aussi une comparaison entre la mise en œuvre proposée et la mise en œuvre utilisant une architecture basée sur un MicroBlaze avec une FPU interne. Nous discutons ensuite de l'ajout d'une unité de contrôle de la FPU dans le but de réaliser des fonctions arithmétiques simple ou complexe. La surface occupée par le module ainsi que ses performances et son extensibilité sont discutées.

3.4.1 État de l'art sur les unités à virgule flottante

Le calcul en virgule flottante est une fonctionnalité incluse dans les processeurs afin de leurs permettre d'effectuer des opérations mathématiques en virgule flottante, ce qui augmente la vitesse et la précision. Dans la communauté libre (open source), les projets sur les FPU sont moins nombreux que ceux sur les processeurs. Toutefois, quelques projets semblent avoir des résultats très intéressants.

Rudolf Usselmann [57] a proposé une FPU simple précision, 32 bits, compatible avec la norme IEEE 754 et supportant huit opérations (addition, soustraction, multiplication, division, conversion de l'entier vers le flottant, conversion du flottant vers l'entier et retenue (les conversion ne sont pas implanter encore)). Les quatre modes d'arrondis proposés par la norme IEEE 754 (vers $+\infty$, vers $-\infty$, vers zéro et au plus près) sont supportés par cette FPU. Cette FPU gère bien aussi le NaN (Not a Number), lors d'une opération invalide telle que 0/0, 0 * ∞ , la sortie sera tout simplement un NaN. Cette FPU est codée en langage verilog et testée avec plus de 14 millions de vecteurs générés à l'aide de la bibliothèque Softfloat [58].
Jidan Al-Eryani [59] a proposé une FPU simple précision, basée sur le même standard IEEE 754. Ce projet supporte cinq opérations (addition, soustraction, multiplication, division et racine carrée) et les quatre modes d'arrondis cité précédemment. Synthétisé sur Altera Quartus II v.5, la fréquence maximum est de 100MHz. Le test de la FPU est fait avec le même outil (SoftFloat) avec environs de 2 millions de vecteurs. L'architecture est implantée sur un FPGA de type Cyclone I–EP1C6.
Marcus Guillermo [60] a proposé une FPU simple précision, compatible avec la norme IEEE 754 et supportant trois opérations (addition, soustraction et multiplication). L'architecture est pipelinée avec six étages pour l'addition et la soustraction et quatre étages

pour la multiplication. L'architecture a été testée à une fréquence de 33MHz sur un Virtex-II XC2V3000.
Pour ce qui est des FPU double précision, Lundgren David [61] a développé une FPU compatible avec la norme IEEE 754 et supportant quatre opérations (addition, soustraction, multiplication et division). Les quatre modes d'arrondis sont supportés par cette FPU. Les temps d'exécution sont de 20 cycles d'horloge pour une opération d'addition, 21 cycles pour une soustraction, 24 cycles pour une multiplication et 71 cycles pour une division. L'architecture peut fonctionner à une fréquence d'horloge de 185MHz sur un Virtex-5. Le tableau 3.3 récapitule les caractéristiques des FPU décrit précédemment dans l'état de l'art.

Afin d'avoir assez d'informations pour choisir la FPU la plus adéquate à notre projet, nous avons évalué les projets présentés précédemment. Une implantation de ces IPs a été effectuée sur cible Virtex-6. Pour tous les résultats, les options d'optimisation lors de la synthèse ont été mis au plus haut niveau de vitesse avec des fortes contraintes temporelles. La Figure 3.3 présente les résultats de synthèse sur cible FPGA Virtex-6 pour les quatre FPUs.

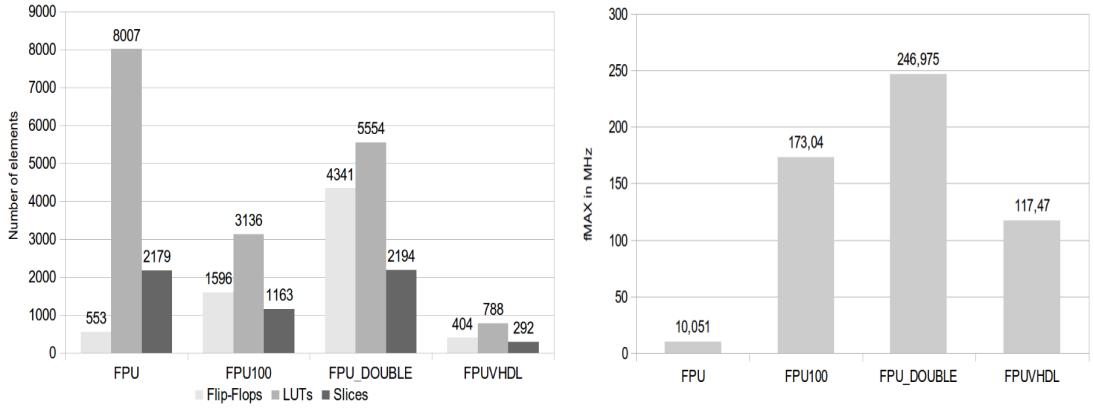


FIGURE 3.3 – Résultats de synthèse sur cible FPGA Virtex-6

Les résultats d'implantations montrent clairement que la FPU100 [59] est la plus performante (compromis vitesse/surface) dans la catégorie des FPUs simple précision. Le choix s'est donc porté sur la FPU100. Cette FPU est la plus intéressante dans la communauté libre, même si ses caractéristiques en termes de surface et vitesse ne sont pas pleinement satisfaisantes. Toutefois, cette FPU nous permet de valider le concept (c'est-à-dire l'ajout d'une FPU avec unité de contrôle) ce qui est l'objectif premier. Vu que le projet est libre, la seule condition pour les utilisateurs souhaitant faire évoluer la FPU est qu'elle doit rester pleinement compatible avec la norme IEEE 754 simple précision.

Nom du Projet sur OpenCores	**fpu [57]**	**Fpu100 [59]**	**fpu_double [61]**	**Fpuvhdl [60]**
Précision	simple précision	simple précision	double précision	simple précision
Norme	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Opérations	add sou mult div int -> float float -> int retenue	add sou mult div racine carrée	add sou mult div	Add sou mult
Modes d'arrondis	vers $+\infty$ vers $-\infty$ vers 0 au plus près	vers $+\infty$ vers $-\infty$ vers 0 au plus près	vers $+\infty$ vers $-\infty$ vers 0 au plus près	-
Langage	Verilog	VHDL	VHDL	VHDL
Test avec SoftFloat	14 millions	2 millions	-	-
Implantation	-	Cyclone I	Virtex 5	Virtex 2
Nombre d'étages de pipeline	4 étages	3 étages	-	4/6 étages
Nombre de cycles	-	Add : 7 sou : 7 mult : 12 div : 35 racine : 35	Add :20 sou : 21 mult : 24 div : 71	-
Compatibilité avec le bus Wishbone	Non	Non	Non	Non

TABLE 3.3 – Récapitulatif des caractéristiques des FPU décrites dans l'état de l'art.

3.4.2 L'unité à virgule flottante choisie

La figure 3.4 montre une vue simplifiée de l'architecture du FPU100, qui a été conçue pour être la plus modulaire possible. Toutes les opérations arithmétiques sont constituées des trois mêmes étages comme présenté sur la figure 3.4. Le premier étage est l'étage *Pré-Normalisateur* où les opérandes sont mis sous un format facilitant leur gestion. Le deuxième étage est l'étage *Cœur arithmétique* où les opérations arithmétiques de base sont effectuées. Enfin, le résultat est normalisé sous le format spécifié

par la norme IEEE dans l'étage *post-Normalisateur*.

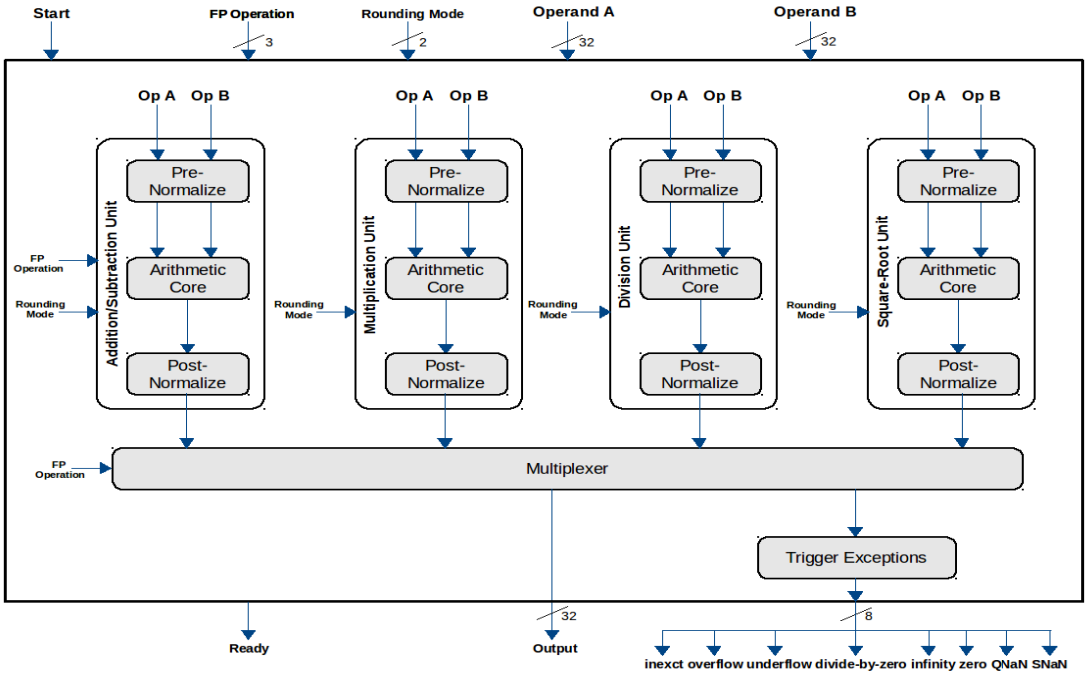


FIGURE 3.4 – Architecture de la FPU100

3.4.3 Intégration de la FPU dans le système

Un de nos principaux objectifs dans ce projet est de fournir au processeur un accès à une FPU sans surcharger l'étage d'exécution. Cette solution présente en plus l'avantage de laisser le processeur libre après la transmission de données à la FPU (ce qui n'est pas le cas pour les processeurs qui intègrent la FPU dans leurs cœurs). Pour ce faire, cinq ports FSL sont connectés à la FPU via les bus FSL. Une machine d'état gère l'état de la FPU (occupé ou prêt à faire un autre traitement). Les détails de l'intégration de la FPU avec des aspects d'optimisations sont présentés dans la l'**annexe A.2**.

3.4.4 Comparaison avec la FPU du MicroBlaze

Une FPU peut être ajoutée au MicroBlaze dans l'étape de configuration du cœur en utilisant le constructeur de système de base BSB (Base System Builder). L'ajout de la FPU dans le cœur du MicroBlaze implique une surcharge du cœur (ressources supplémentaires). Les tableaux 3.4 et 3.5 représentent les ressources utilisées dans le cas d'un MicroBlaze avec et sans FPU et dans le cas d'un SecretBlaze avec et sans FPU. Les architectures ont été implantées sur le Virtex-6.

Comme le montre le tableau 3.4, il y a une augmentation significative dans les ressources matérielles utilisées entre le MicroBlaze sans FPU et le MicroBlaze avec FPU.

	MicroBlaze sans FPU		MicroBlaze avec FPU	
	Surface non optimisée	Surface optimisée	Surface non optimisée	Surface optimisée
Nombre de Flip-Flops	1567 (0,52%)	1280 (0,42%)	2290 (0,76%)	1998 (0,66%)
Nombre de LUTs	1562 (1,04%)	1230 (0,82%)	2771 (1,84%)	2350 (1,56%)

TABLE 3.4 – Résultats d'implantation sur MicroBlaze

Dans le cas d'optimisation de surface, les ressources utilisées ont augmenté de $56,09\%$ en Flip-flops et $91,05\%$ en LUT. Pour le cas où la surface n'est pas optimisée, les ressources utilisées ont augmenté de $46,13\%$ en Flip-flops et $77,40\%$ en LUT. Dans la version où la surface est optimisé, les instructions en virgule flottante prennent deux cycles de plus par rapport à la version où la surface n'est pas optimisée.

	SecretBlaze sans FPU	SecretBlaze avec FPU
Nombre de Flip-Flops	913 (0,30%)	2569 (0,85%)
Nombre de LUTs	1677 (1,11%)	4988 (3,31%)

TABLE 3.5 – Résultats d'implantation sur SecretBlaze

L'augmentation significative des ressources matérielles du SecretBlaze avec FPU (tableau 3.5) est due au faite que la FPU choisie occupe des ressources importantes par rapport à l'ensemble du système (1596 Flip-Flops soit $62,12\%$ de l'ensemble du système et de 3136 LUT soit $62,87\%$ de l'ensemble du système).
Pour évaluer les performances de la solution proposée, le tableau suivant (tableau 3.6) indique les temps de traitement (en microsecondes) pour un calcul vectoriel de deux vecteurs de dimension 256 pour 4 opérations (addition, soustraction, multiplication et division) en utilisant le SecretBlaze sans FPU (approche logicielle) et en utilisant le SecretBlaze avec FPU par FSL (solution proposée).

Les résultats montrent que la solution proposée est environ cinq fois plus rapide que la solution avec SecretBlaze sans FPU. La solution proposée est plus lente que la solution de MicroBlaze avec FPU mais cela pourrait s'expliquer par les performances modeste de la FPU choisie (qui est pourtant la plus performante trouvée sur la communauté libre). Sachant que les instructions FSL ne sont pas un handicap comme décrit ci-dessus, il faut noter aussi que même si les résultats de MicroBlaze avec FPU sont meilleurs, notre architecture permet de libérer le processeur afin d'exécuter d'autres calculs en parallèle.

	MicroBlaze		SecretBlaze	
	sans FPU (surface non optimisée)	avec FPU (surface non optimisée)	sans FPU	avec FPU
add	527,89	41,51	349,72	69,07
sub	499,73	41,51	258,23	69,07
mult	635,40	41,51	297,66	75,47
div	1155,09	102,95	867,99	127,39

TABLE 3.6 – Temps d'exécution pour les opérations vectorielles(μs)

3.4.5 L'unité de contrôle FPU

Afin d'améliorer les performances et maintenir le processeur aussi libre que possible lorsque la FPU est en cours de calcul, une unité de contrôle de la FPU a été développée et ajoutée à l'architecture avec l'intention de mettre en œuvre plusieurs fonctions mathématiques spécifiques. L'unité de commande de la FPU est connectée au processeur. Elle reçoit les données et la fonction à réaliser par le processeur. Elle contrôle la FPU (voir la figure 3.5) selon la fonction sélectionnée. Une fois le calcul terminé, une requête d'interruption permet d'indiquer que le résultat est prêt afin de le transmettre au processeur. La figure suivante (fig. 3.5) montre l'ensemble de l'architecture du système. L'unité de commande FPU est connectée au processeur via le bus FSL de sorte que de multiples fonctions peuvent être envoyées successivement tandis que les résultats peuvent être lus à tout moment.

3.4.5.1 Méthode d'implantation de fonctions spécifiques

L'approximation polynomiale est la méthode utilisée pour implanter les fonctions mathématiques complexes. L'approximation d'une fonction par un polynôme est une solution largement répandue pour représenter n'importe quelle fonction, simple ou complexe (comme les fonctions trigonométrique (sinus, cosinus,) ou les fonctions logarithmique (log, ln, ...)). Plusieurs architectures matérielles utilisent l'approximation polynomiale comme moyen de réaliser des fonctions mathématiques. ADSP-218x [62] de Analog Devices [63] est l'une de ces architectures qui utilise l'approximation pour réaliser des fonctions. Les polynômes sont des fonctions simples qui peuvent être construits en utilisant seulement 3 opérations +, -, *. La façon la plus simple de représenter un polynôme (P) de degré inférieur ou égal à n, est d'utiliser la base canonique 1, x, x ^{2}, .. x ^{n} (équation 3.1). Toutefois, ce processus peut être appliqué de façon récursive pour obtenir une fonction simplifiée (équation 3.2).

$$p(x) = \sum_{i=0}^{i=n} a_i.x^i \tag{3.1}$$

$$p(x) = a_0 + x.(a_1 + x.(a_2 + x.(a_3 + ... + x.(a_{n-1} + x.a_n)...)) \tag{3.2}$$

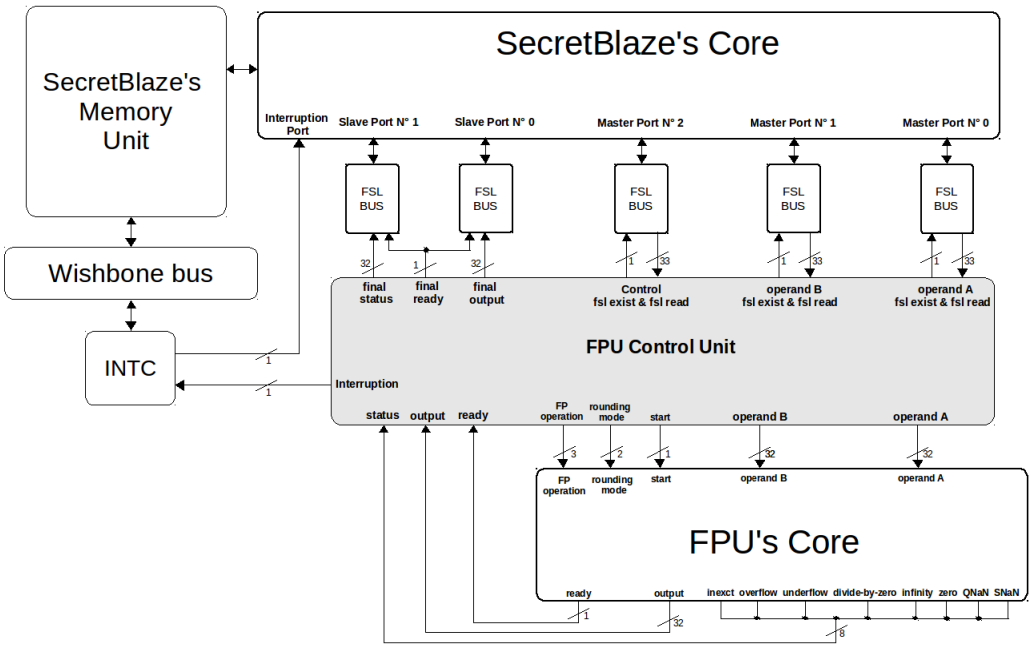


FIGURE 3.5 – Vue d'ensemble de l'architecture du système

Les coefficients sont calculés grâce à des outils mathématiques comme ORIGIN [64]. Ces outils fournissent une approximation selon l'ordre choisie par l'utilisateur. Avec un polynôme d'ordre 5 (ordre choisie dans cette implantation), l'approximation donne un écart type très faible.

3.4.5.2 Exemples de fonctions mises en œuvre

Trois fonctions ont été mises en œuvre dans ce projet à titre d'exemple :

Exemple 1 : (x op y) Cette fonction est implantée dans l'unité de contrôle de la FPU dans le but de maintenir les cinq opérations standard (détaillées ci-dessus) et de gagner des coûts d'horloge. Cette fonction est plus rapide que le cas particulier de la fonction $(x\ op\ y)^1$.

Exemple 2 : $(x\ op\ y)^i$ Cette fonction est mise en œuvre dans l'unité de contrôle de la FPU. Les calculs possibles sont $(x + y)^i$, $(x - y)^i$, $(x * y)^i$, $(x / y)^i$. La fonction de puissance est très consommatrice en terme de temps d'exécution. Les résultats obtenus avec l'approche proposée montre que le temps d'exécution sans l'unité de contrôle de la FPU est 10 fois plus grand que le temps d'exécution avec l'unité de contrôle de la FPU. Par exemple, le temps d'exécution de $(x + y)^{32}$ en utilisant la solution logicielle est de 59μs, tandis qu'il est seulement de 5,9 μs avec l'unité de contrôle de la FPU.

Exemple 3 : sinus Dans le domaine du traitement d'image et du signal, l'utilisation des fonctions trigonométriques est très courant. Dans cette exemple d'implantation (fonction sinus), le polynôme d'approximation suivant a été utilisé afin de représenter la fonction sin(x) (la variable x représente l'angle à calculer exprimé en radian).

$$\sin(x) = 0.999691986.x + 0.002053139.x^2 - 0.171745742.x^3 + 0.005591653.x^4 + 0.005882932.x^5 \quad (3.3)$$

L'ordre du polynôme d'approximation utilisé est égale a 5, ce qui est très suffisant pour n'importe quelle valeur de x comprise entre 0 et $\pi/2$. En utilisant des formules trigonométriques comme sin(-x) = -sin(x) et sin(x) = sin(π - x), n'importe quelle angle peut ainsi être calculé.

Les coefficients du polynôme sont initialisés dans la machine d'état *SIN FSM*. Une comparaison a été effectuée entre le calcul d'un sinus en utilisant des librairies logicielles (le compilateur utiliser est mb-gcc de chez Xilinx) et un calcul de sinus en utilisant l'unité de contrôle FPU. Les résultats montrent que le temps d'exécution sans l'unité de contrôle FPU est 3 fois plus grand que le temps d'exécution en utilisant l'unité de contrôle FPU. (1.3μs avec l'unité de contrôle FPU et 3.9μs avec les librairies logicielles).

3.4.5.3 L'architecture de l'unité de contrôle FPU

L'unité de contrôle FPU a été conçue en vue de flexibilité, d'évolutivité et de facilité au niveau du logiciel et aussi une utilisation facile pour l'ajout de nouvelles fonctions du point de vue utilisateur. L'unité de contrôle FPU contient deux types de modules : les modules génériques qui sont utilisés pour toutes les fonctions et les modules spéciaux qui sont utilisés pour mettre en œuvre une fonction spécifique.

Les modules génériques : Il existe trois modules génériques : *INOUT CONTROL*, *FUNCTION FSM* and *STATE FSM* comme le montre la figure 3.6 :

• *STATE FSM* est une machine d'état connectée au bus FSL et à la *FUNCTION FSM* permettant de contrôler les données dans le bus FSL et l'état de la FPU (occupée ou disponible). En effet, si la FPU est prête à exécuter à un nouveau calcul, *STATE FSM* lit le bus FSL.

• *FUNCTION FSM* est aussi une machine d'état permettant de sélectionner la fonction à exécuter. Il permet de connecter la fonction en cours à la *STATE FSM* et au *INOUT CONTROL*.

• *INOUT CONTROL* est module implanté pour gérer le flux de données. Il reçoit les opérandes A et B, l'opération à exécuter et le mode d'arrondi, soit directement à partir du processeur soit de la fonction en cours d'exécution. Dans le cas où les données

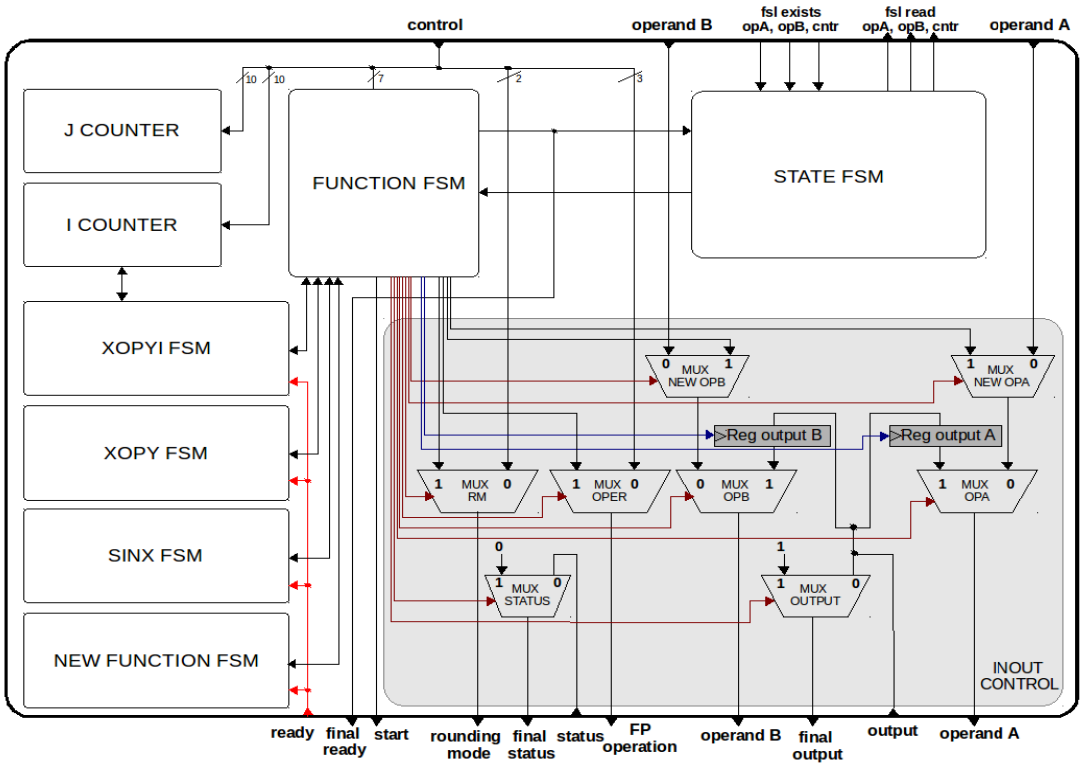


FIGURE 3.6 – Architecture de l'unité de contrôle FPU

viennent de la FPU, cela signifie que le résultat de calcul est utilisé comme une entrée pour le prochain calcul (cas des fonctions récursives). Ce module est très flexible comme le montre la figure 3.6.

Les modules spéciaux : Ce sont les modules qui permettent la description et la réalisation de la fonction mathématique à implanter :

• *I COUNTER et J COUNTER* : la nécessité d'un ou plusieurs compteurs dépend de la fonction à traiter. Par exemple, pour une fonction qui utilise deux boucles imbriquées, l'utilisation de deux compteurs facilite la conception de la machine d'état de la fonction.

• *XOPY FSM, XOPYI FSM and SINX FSM* : ces trois modules (machines d'états) sont les trois fonctions mises en œuvre dans ce projet afin d'illustrer le concept. Pour réaliser la fonction *XOPY*, une machine d'état unique est suffisante pour décrire la fonction (de même que pour la fonction *SINX*). La fonction *XOPYI* nécessite une machine d'état et un compteur afin de gérer la puissance.

Le port de contrôle (fig. 3.6) reçoit les données à partir du port maître N^o 2 (voir le lien sur la figure 3.5). Ce signal donne des informations sur l'opération. Les cinq composants de base sont l'opération (add, sub, mult, div, squ), mode d'arrondi (arrondi : le plus proche, à zéro, haut, bas), fonction (xopy, xopyi, sin, etc ...), la valeur d'itération I (10 bits) et la valeur d'itération J (10 bits). La disposition de ces informations pour le signal de contrôle a été définie pour anticiper les futures fonctions. Cependant, cette structure peut être modifiée par la suite et adaptée aux besoins des utilisateurs, sachant que la complexité de la machine d'état des fonctions diminuera avec une structure efficace du signal de contrôle.

3.4.5.4 Résultats d'implantation

Le tableau 3.7 montre les ressources utilisées dans le cas d'une architecture supportant deux fonctions (x op y) and $(x\ op\ y)^i$. Le tableau 3.8 montre les ressources utilisées dans le cas d'une architecture supportant trois fonctions (x op y), $(x\ op\ y)^i$ et sin(x).

Modules	Supportant 2 fonctions (4 FSMs)		
	Slice Reg	LUTs	Slices
FPU CONTROL UNIT	103 (0,19%)	136 (0,50%)	84 (1,23%)
INOUT CONTROL	64 (0,12%)	96 (0,35%)	64 (0,94%)
I COUNTER	10 (0,02%)	13 (0,05%)	6 (0,09%)
FSMs : STATE, FUNCTION, XOPY, XOPYI	29 (0,05%)	27 (0,10%)	14 (0,20%)

TABLE 3.7 – Résultats d'implantation dans le cas d'une architecture supportant 2 fonctions)

Les deux tableaux montrent que l'implantation d'une fonction sinus fait augmenter les ressources utilisés de 25 slice reg $(0,05\%)$ et de 23 LUTs $(0,08\%)$. La surface ajoutée est négligeable même avec un FPGA de taille modeste comme celui utilisé dans ce projet (XC6SLX45). Des résultats équivalents peuvent être attendus pour d'autres fonctions trigonométriques (comme : cosinus, tangent,...) implantées avec un polynôme de même ordre.

3.4.5.5 Comment ajouter des nouvelles fonctions ?

L'architecture de l'unité de contrôle FPU est très flexible et peut supporter n'importe quelle fonction. Pour ajouter une nouvelle fonction, la seule chose à faire par l'utilisateur est la création d'une machine d'état qui décrit la fonction, puis la connecté comme illustrée dans la figure 3.6.

Modules	Supportant 3 fonctions (5 FSMs)		
	Slice Reg	LUTs	Slices
FPU CONTROL UNIT	128 (0,23%)	159 (0,58%)	95 (1,39%)
INOUT CONTROL	64 (0,12%)	96 (0,35%)	62 (0,91%)
I COUNTER	10 (0,02%)	13 (0,05%)	6 (0,09%)
FSMs : STATE,FUNCTION, XOPY, XOPYI, **SINX**	54 (0,10%)	50 (0,18%)	27 (0,39%)

TABLE 3.8 – Résultats d'implantation dans le cas d'une architecture supportant 3 fonctions)

Nous avons présenté l'ajout d'un module pour le calcul en virgule flottante. Le système est libre, flexible et évolutif. L'unité de contrôle FPU proposée permet d'ajouter facilement et rapidement des fonctions complexes bien adaptées à l'application visée. Les résultats obtenus sur cible FPGA, ont montré que les fonctions ajoutées présentes des performances très intéressantes en terme de temps d'exécution par rapport à un calcul logiciel, avec une modeste augmentation de surface (25 FF et 23 LUT pour une fonction sinusoïdale estimés avec un polynôme à 5 ^{ieme} ordre). Au niveau logiciel , les fonctions sont faciles à utiliser et occupent beaucoup moins d'espace dans la mémoire. Un autre aspect de ce travail est que le processeur reste libre de traiter d'autres fonctions quand la FPU est en cours de traitement.

3.5 L'organisation de la mémoire

Après avoir présenté le processeur et les améliorations que nous avons apportées, nous allons discuter de la mémoire qui va être implantée dans le nœud. Les approches d'organisation de la mémoire ont été discutées dans le chapitre 1 et 2. Et pour rappel, le modèle d'organisation de la mémoire choisi est de type MIMD-DM (Multiple Instruction Multiple Data-Distributed Memory). Étant donné que la méthodologie HNCP est basée originellement sur une cible FPGA, les mémoires utilisées sont de type RAM double ports, ce qui correspond au type de processeur et aux modules implantés utilisant la mémoire. La mémoire doit être accessible par le processeur en utilisant tout le port assurant ainsi une bande passante totale, afin de garder une performance maximale du processeur et ne pas tombé dans le piège du modèle SM (Shared Memory). Les autres modules doivent prendre l'autre port tel qu'est le cas du DMA-Routeur. Cette contrainte ne pose pas de problème pour ce qui est de notre passage de FPGA vers ASIC, car les mémoires de type RAM double ports sont disponibles avec différentes configurations pour la cible ASIC.

3.6 Ajout du JTAG : un moyen de programmation et de debug

L'architecture globale du système JTAG (Joint Test Action Group) proposée est basée sur le standard IEEE Std. 1149.1-1990 [65]. Le module permet d'agir sur des parties (*Design Module*) du circuit implanté dans la cible (ASIC ou FPGA) comme le montre la figure 3.7. Le module permet de faire un contrôle du fonctionnement, une observation d'états, une lecture dans des zones mémoire et bien sûr une écriture dans des zones mémoire (chargement d'un programme). Ce dernier point est la principale motivation à la réalisation de ce module.

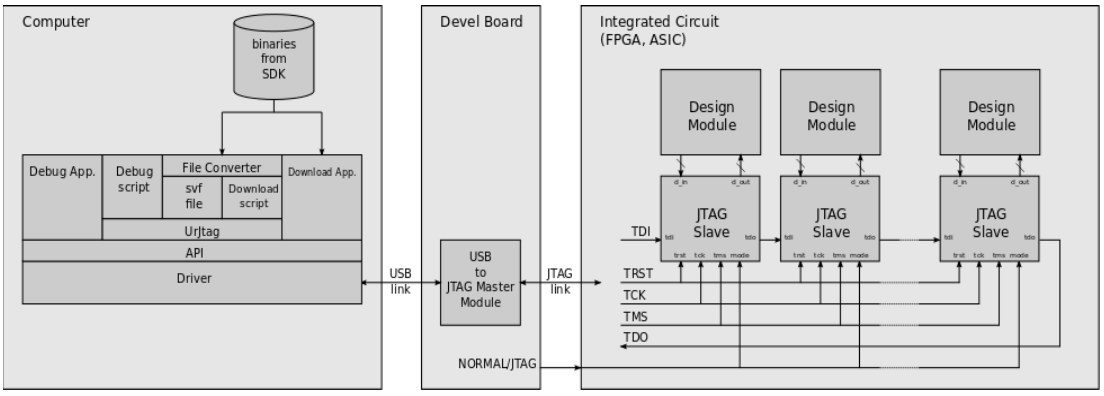


FIGURE 3.7 – Architecture générale du système JTAG.

Chaque module est associé à un module JTAG esclave (détails de l'architecture dans l'**annexe A.3**) inséré dans une chaîne rassemblant tous les esclaves. La chaîne est accessible depuis l'extérieur du circuit par l'intermédiaire d'un port JTAG. Une broche de configuration permet de définir le mode de fonctionnement de la chaîne JTAG. Nous avons un mode normal où les esclaves sont déconnectés de leur module associé (l'ensemble des cellules présentes dans les chaînes "Jtag Data Registers" deviennent transparentes) et nous avons un mode JTAG où les esclaves sont connectés à leur module associé (lecture/programmation des mémoires et lecture/écriture des registres de configuration).
Le lien JTAG connecté au port JTAG du circuit est piloté par un module maître (*USB to JTAG Master Module*) implanté sur la carte électronique (*Devel Board*) accueillant le circuit (fig. 3.7). Le module maître est contrôlé à travers une liaison USB par une application s'exécutant sur un ordinateur (*Computer*)(fig. 3.7). Le pilote (*Driver*) permet d'accéder au module maître. Les fonctions de communication liées à ce pilote sont accessibles à l'utilisateur à travers une interface de programmation API (Application Programming Interface). L'application utilisateur, comporte deux fonctions principales : la première consiste dans la programmation des mémoires à partir des fichiers binaires générés par la chaîne de développement logicielle (SDK (Software Development Kit)), La deuxième consiste dans le debug du circuit.

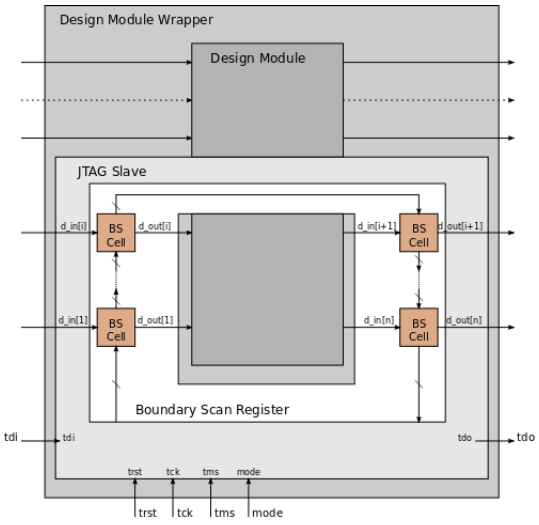


FIGURE 3.8 – Le système JTAG esclave associé à un module

3.6.1 Résultats d'implantations

Le tableau 3.9 présente l'utilisation des ressources matérielles du JTAG sur la cible Virtex-6 avec différentes longueur de la chaîne JTAG. Comme le montre le tableau, il y a une forte augmentation des ressources matérielles utilisées chaque fois que la largeur du registre augmente. Ce qui est évident, car la longueur de la chaîne JTAG, dépend du nombre de cellules du registre *boundary scan*.

Longueur de la chaîne JTAG	2	32	64	**103**	128	256
Nombre de Flip-Flops	68 (0,02%)	123 (0,04%)	187 (0,06%)	**271 (0,09%)**	322 (0,11%)	610 (0,20%)
Nombre de LUTs	57 (0,04%)	161 (0,11%)	272 (0,18%)	**408 (0,27%)**	497 (0,33%)	949 (0,63%)
Nombre de Slices	41 (0,11%)	73 (0,19%)	141 (0,37%)	**184 (0,49%)**	219 (0,58%)	402 (1,07%)

TABLE 3.9 – Résultats d'implantations

Afin d'illustrer l'utilisation des ressources matérielles du JTAG pour la programmation du SecretBlaze, 103 bits sont nécessaires pour contrôler la mémoire via son port de données relier au cœur du processeur. 32 bits d'adresses, 32 bits de données d'entrées, 32 bits de données de sorties, 4 bits pour le signal de sélection, 1 bit pour le signal d'écriture/lecture, 1 bit pour le signal enable et 1 bit pour le signal d'horloge.

3.7 Protocoles de communication

Deux protocoles de communication seront utilisés donc notre projet, le premier est de type point à point (FSL), déjà utilisé dans la méthodologie HNCP, le deuxième est le bus Wishbone.

3.7.1 Protocole de communication par bus FSL

Le bus FSL [56] est un moyen rapide de communication entre le processeur et une autre entité. Chaque lien FSL 32 bits est unidirectionnel, point-à-point et met en œuvre une FIFO et des signaux de contrôle. La figure 3.9 montre les signaux maître/esclave utilisés lors d'une opération de lecture ou d'écriture.

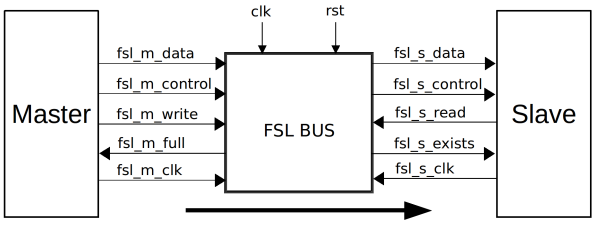


FIGURE 3.9 – Interfaces maître/esclave du bus FSL

3.7.2 Protocole de communication par bus Wishbone

Le bus Wishbone [55] est un bus libre destiné à permettre à différentes IP de communiquer entre elles. Un très grand nombre de conception libre ont adapté leur architecture à l'interface Wishbone. La figure 3.10 montre les signaux maître/esclave utilisés lors d'une connexion avec le bus Wishbone. Certains signaux utilisés dans la connexion Wishbone sont optionnels et peuvent être ou ne pas être présents sur une interface spécifique.
Le SecretBlaze est l'un des projets libre qui utilise le bus Wishbone comme bus d'interconnexion. Dans le SecretBlaze le bus Wishbone sert à connecter le cœur du processeur au périphériques d'entrées/sorties comme UART, GPIO, Timer, ..etc. L'utilisation de ce bus va être la même dans le cas de notre architecture, c'est-à-dire que le Wishbone va servir uniquement de connexion entre le processeur et les périphériques.

3.7.3 Résultats d'implantation

Le tableau 3.10 présente l'utilisation des ressources matérielles des implémentations des bus FSL et Wishbone utilisant le Virtex-6. Le bus Wishbone est configuré pour supporter deux maîtres (deux interfaces : instructions et données) et cinq esclaves (périphériques : GPIO, UART,...). Le bus FSL est configuré avec une profondeur de FIFO égale à 16 afin d'assurer le fonctionnement en mode asynchrone (16 étant la profondeur minimale pour assurer le mode Asynchrone du Bus).

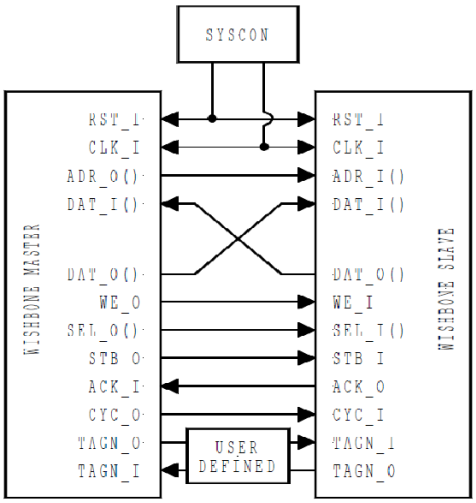


FIGURE 3.10 – Interfaces maître/esclave du bus Wishbone [55]

Bus	Bus FSL	Bus Wishbone
Nombre de Flip-Flops	14 (0,005%)	5 (0,002%)
Nombre de LUTs	35 (0,02%)	67 (0,04%)
Nombre de Slices	17 (0,04%)	42 (0,11%)

TABLE 3.10 – Résultats d'implantation pour les bus FSL et Wishbone

3.8 Le DMA-Routeur

Dans cette section, nous allons introduire et définir une solution de gestion des communications basée sur l'utilisation d'un Routeur et d'un DMA (Direct Memory Access). Dans la version originale du projet, seule la topologie hypercube est implantée. Après plusieurs thèses et stages travaillant sur ce module, celui-ci a atteint un degré de généricité facilitant ainsi son insertion dans la méthodologie HNCP [25]. Nous nous intéressons dans un premier temps au fonctionnement et à l'aspect matériel de ce module. Nous illustrons par la suite les modifications que nous avons apportées afin qu'il soit adapté à deux autres topologies qui sont la grille et le tore. Nous présentons par la suite une évaluons de ce module. Enfin, nous présentons les fonctions logicielles nécessaires à son utilisation.

3.8.1 Principe de fonctionnement du DMA-Routeur

Le rôle principal du DMA est de permettre l'accès direct à la mémoire sans passer par le processeur, permettant ainsi de diminuer la charge des communications sur le processeur. Dans le cas d'une émission, le DMA effectue une lecture en mémoire selon la configuration reçue par le processeur, met en forme les paquets, puis transmet les paquets au Routeur. Dans le cas d'une réception, le DMA reconstitue les données à partir des paquets reçus par le Routeur et écrit les données en mémoire selon la configuration reçue par le processeur.

Pour ce qui est du Routeur, le protocole de communication utilisé est celui de Wormhole [66], qui est un protocole de communication par passage de messages. Un message est un ensemble de paquets. Un paquet de taille **L** (paramétrable) est constitué de deux flits d'entêtes (flits (**fl**ow control dig**its**)), suivis de **(L-2)** flits de données de 32 bits. La figure 3.11 montre la représentation d'un paquet et le contenu de ces deux entêtes. Dès qu'un flit est bloqué au niveau d'un nœud, les flits cessent de transiter et restent stockés dans les nœuds où ils se trouvaient (selon l'espace tampon réservé). L'émission des paquets reprend dès que le blocage est levé. Comme il a été mentionné précédemment, la topologie implantée est l'hypercube dont la fonction de routage n'est qu'une simple porte logique de type OU exclusif (XOR). Afin d'éviter l'inter-blocages [67] (deadlock), les concepteur du routeur ont utilisé l'algorithme de routage de [67]. Basés sur l'utilisation de canaux virtuels, ceux-ci permettent de stocker les flits d'une manière transparente dans les buffers.

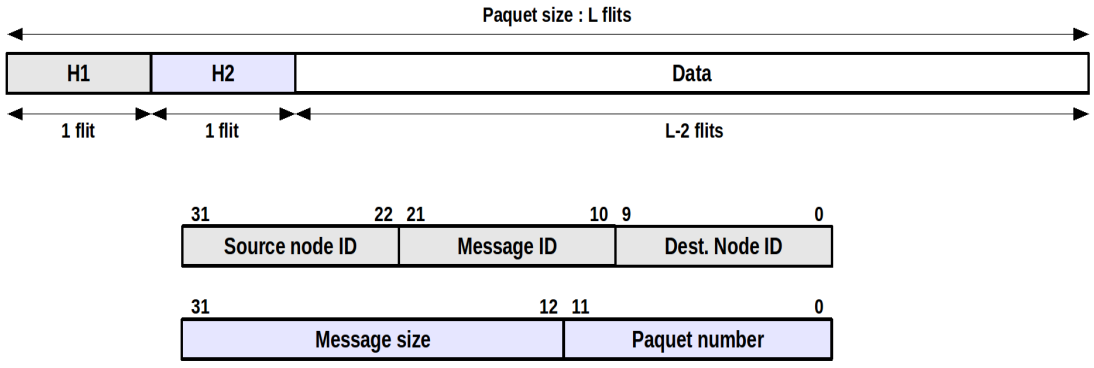


FIGURE 3.11 – Représentation d'un paquet et le contenu de ces entêtes.

3.8.2 Architecture du DMA-Routeur

La figure 3.12 présente l'architecture du DMA-Routeur. Ce module est constitué de deux blocs fonctionnels principaux : Le DMA (cadre rouge du dessus) et le routeur (cadre rouge du dessous). Le module DMA est connecté à deux mémoires, mémoire

d'émission et mémoire de réception, les deux accessibles par le processeur. Il est connecté aussi au cœur du processeur via une liaison point-à-point de type FSL. Le routeur quant à lui est connecté au réseau (les autres routeurs du nœud) via des canaux constitués de 33 bits (32 bits de données et 1 bit pour le canal virtuel), 1 bit request et 1 bit Acknowledge. Pour assurer une communication bidirectionnel (émission et réception) entre deux routeurs, il faut 2 x 35 liens.

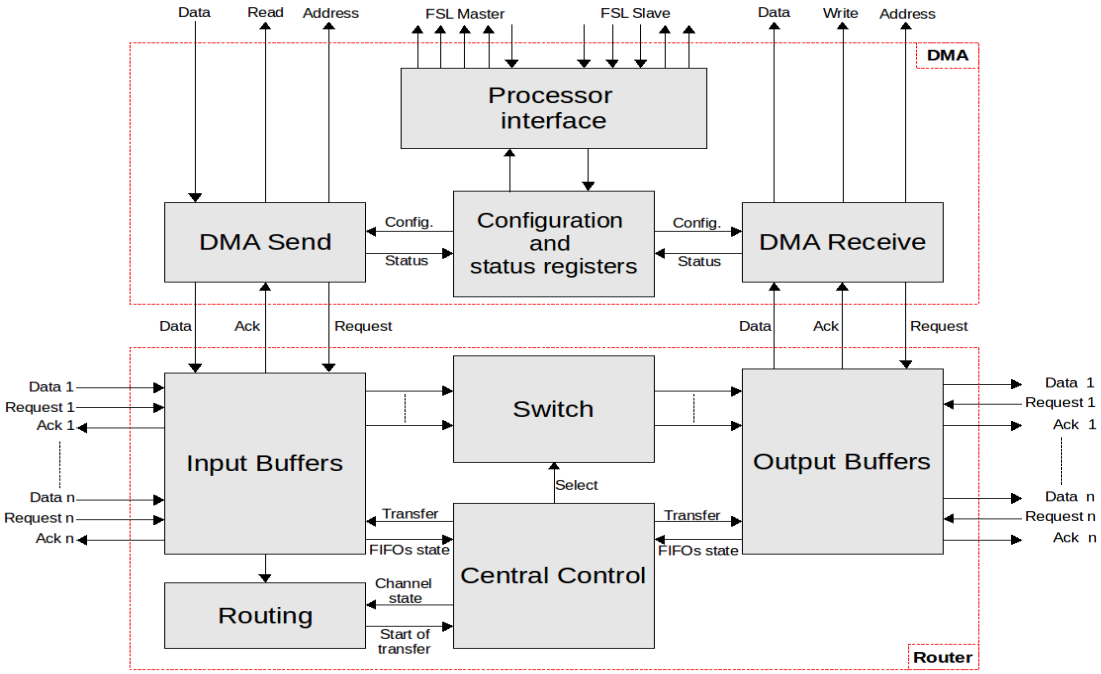


FIGURE 3.12 – Architecture du DMA-Routeur.

3.8.2.1 Architecture du DMA

Le module DMA est composé de quatre sous-modules (fig. 3.12) :

• **Processor interface** : est une machine d'état qui gère les communications avec le processeur. Ce module décode les commandes transmises par le processeur et met à jour les registres de configuration. La communication entre ce module et le processeur est assuré via des liens FSL.

• **Configuration and status registers** : Ce sous-module contient les registres de configuration qui servent à stocker les informations nécessaires pour effectuer une émission ou une réception, tel que le numéro du nœud destinataire, la taille du message...etc. Quatre registres sont dédiés à l'émission et quatre registres sont dédiés à la réception, ce qui fait quatre configurations possibles pour chaque type de communication.

• **DMA Send** : est une machine d'état connectée à la mémoire d'émission et aux registres de configuration. Elle est connectée aussi aux *Input Buffers*. Ce module lit les

données de la mémoire, met en forme les paquets en insérant les deux flits d'entêtes et transmet les paquets au module *Input Buffers*.
• **DMA Receive** : est une machine d'état connectée à la mémoire de réception et aux registres de configuration. Elle est connectée aussi aux **Output Buffers**. Ce module récupère les données du module *Output Buffers*, supprime les paquets d'entêtes et stocke les données reçues dans la mémoire de réception.
L'architecture du DMA assure des canaux full-duplex, grâce à l'indépendance totale entre les deux modules *DMA Send* et *DMA Receive*, ce qui fait que ces deux modules peuvent fonctionner simultanément.

3.8.2.2 Architecture du Routeur

Le module Routeur est composé de cinq sous-modules (fig. 3.12) :
• **Input Buffers** : est un ensemble de registres qui permettent le stockage temporaire des flits. Chaque canal comporte un buffer (approche de type FIFO) y compris le canal qui relit ce module au module *DMA Send*.
• **Output Buffers** : est aussi un ensemble de registres qui permettent le stockage temporaire des flits. Chaque canal comporte deux buffers (approche de type FIFO) y compris le canal qui relit ce module au module *DMA Receive*.
• **Routing** : ce sous-module permet de décoder les entêtes des paquets entrants. C'est ce module qui permet d'implanter la topologie souhaitée.
• **Central Control** ce sous-module permet de pilote le sous-module *Switch* et de contrôler l'ensemble des transferts de données à l'intérieur du routeur.
• **Switch** : ce sous-module a une fonctionnalité de multiplexage (crossbar). Il reçoit la configuration du sous-module *Central Control* et effectuer le routage.

3.8.3 Ajout de deux nouvelles topologie : grille et tore

Dans le chapitre 1, nous avons mis en cause la topologie utilisée dans la méthodologie HNCP au même ordre que la cible FPGA. De ce fait et dans le cadre de cette thèse, nous avons ajouté deux nouvelle topologies qui sont : la grille et le tore. Comme il a été mentionné lors de la description de l'architecture du Routeur, celui-ci contient le module *Routing* responsable du protocole de routage. L'ajout des deux topologies nécessite la mise en œuvre d'un algorithme de routage pour chaque type de topologie. Des exemples de pseudo-codes de routage pour les topologies grille et tore sont présentés en annexe **annexe A.4**.

3.8.4 Ré-usinage du code

Le code du DMA-Routeur, qui nous a été fourni, est synthétisable et fonctionnel. Mais, ce code est le résultat d'un travail de plusieurs équipes durant plusieurs années. Chacune de ces équipes a ces propres méthodes d'écriture, ce qui a rendu le code non homogène et difficile à maintenir, engendrant ainsi des problèmes de fonctionnement

et d'utilisation dans certaines applications. C'est pour cela que nous avons entamé une phase de ré-usinage (Refactoring) du code avant d'effectuer les modifications. Le ré-usinage de code est basé sur un ensemble de règles que nous avons mis en place. Allant de l'organisation du projet, passant par les nom des fichier et des modules jusqu'à la mise en page et les commentaires, ... Ce travail a facilité la modification du code : ajout des topologies et optimisation du code. Durant cette phase de ré-usinage, une partie du code a été réécrite, optimisant ainsi les ressources matérielles, ce qui peut être vérifié dans la section suivante. Le projet (code source, testbenchs et datasheet) est disponible en libre accès sur le site de l'équipe DREAM [68] de l'Institut Pascal.

3.8.5 Évaluation des performances

Pour un réseau hypercube utilisant le protocole Wormhole sans congestion [25], la bande passante du réseau s'exprime de la façon suivante :

$$B_w = \frac{4F \times (L-2)}{(L-2) + end_{lat}} \tag{3.4}$$

Avec F étant la fréquence du système, L la taille des paquet et end_{lat} la latence de fin de transmission.

Le réseau complet a été testé sur le Virtex-6 avec comme fréquence d'horloge du système 100MHz. La taille des paquets est configurée à $L = 32$, ce qui donne une bande passante de 342Mo/s.

Estimation du temps de communication : Le temps de communication peut être évalué en fonction du diamètre qui sépare l'émetteur et le récepteur. L'estimation du temps total peut être exprimée comme suit :

$$T_{total} = T_{config} + T_{fdim} + T_{fdat} + T_{mem} \tag{3.5}$$

avec :
T_{config} : temps nécessaire à la configuration d'émission pour la lecture de la mémoire = 20 cycles d'horloge
T_{fdim} : temps nécessaire en fonction du diamètre (variable)
T_{fdat} : temps nécessaire en fonction du nombres des données (variable)
T_{mem} : temps nécessaire pour l'écriture dans la mémoire = 11 cycles d'horloge

Le T_{fdim} temps (en cycle d'horloge) en fonction du diamètre qui sépare l'émetteur et le récepteur peut être exprimé comme suit :

$$T_{fdim} = (diamètre - 1) \times 4 \tag{3.6}$$

Le T_{fdat}, temps (en cycle d'horloge) en fonction de la quantité de données à transmettre peut être exprimée comme suit :

$$T_{fdat} = (diamètre) \times 4 \tag{3.7}$$

Par exemple, soit une architecture de dimension 4 : Entre le nœud 0 et le nœud 15, le diamètre est égal à 4. Le temps (en cycle d'horloge) de transmission d'une donnée est : T = 20 + 12 + 11 + 4 = 47.

Le tableau 3.11 présente le temps T_{fdim} en nombre de cycles en fonction du diamètre qui sépare l'émetteur et le récepteur :

Diamètre	1	2	3	4	5	6
T_{fdim} (cycles d'horloge)	0	4	8	12	16	20

TABLE 3.11 – Estimation du temps T_{fdim} en fonction du diamètre

Le tableau 3.12 présente le temps T_{fdat} en nombre de cycles en fonction du nombre de données (32 bits chacune) à transmettre :

Nombre de données	1	2	3	4	5	6
T_{fdat} (cycles d'horloge)	4	8	12	16	20	24

TABLE 3.12 – Estimation du temps T_{fdat} en fonction du nombre de données

Le tableau 3.13 présente le temps de communication (T_{total}) d'une donnée en fonction du diamètre :

Diamètre	1	2	3	4	5	6
T_{total} (cycles d'horloge)	35	39	43	47	51	55

TABLE 3.13 – Estimation du temps (T_{total}) d'une donnée en fonction du diamètre

3.8.6 Résultats d'implantation

Le tableau 3.14 présente l'utilisation des ressources matérielles dans l'implantation des topologies hypercube, grille et tore utilisant le Virtex-6. Le seul point qui diffère entre les trois topologies est le module *Routing*. Selon la topologie choisie, un code sélectionné est synthétisé, le reste est ignoré. Nous pouvons observer que le code de l'algorithme de routage de la topologie grille occupe les mêmes ressources que celui de la topologie tore. Nous pouvons remarquer aussi la faible utilisation des ressources pour la topologie hypercube (dimension égale à 4) est attendu du au fait que le code est un OU exclusif (XOR).

topologie	Grille	Tore	Hypercube
Nombre de Flip-Flops	1477 (0,49%)	1477 (0,49%)	681 (0,27%)
Nombre de LUTs	1380 (0,92%)	1380 (0,916%)	970 (0,64%)
Nombre de Slices	728 (1,93%)	728 (1,93%)	386 (1,02%)

TABLE 3.14 – Résultats d'implantation pour les trois topologies

3.8.7 Les fonctions logiciels du DMA-Router

Le DMA-Routeur est fourni avec un pilote (driver) constitué de fonctions écrites en langage C. Ces fonctions sont basées sur les instructions FSL. Dans le cadre de cette thèse, nous avons adapté le pilote du DMA-Router avec les instructions FSL du Secret-Blaze, car dans la version originale de celui-ci, les instructions utilisées étaient celles du MicroBlaze. Le pilote est constitué de cinq fonctions : une pour la configuration en émission (*dma_snd_config*), une pour la configuration en réception (*dma_rcv_config*), une pour l'émission (*dma_snd*) , une pour l'attente et la réception (*dma_wait_for_rcv*) et une fonction de nettoyage (réinitialisation) des registres (*dma_clear_rcv_fla*). Deux exemples de fonctions de configuration sont présentés en annexe **annexe A.5**.

L'idée était de permettre à l'utilisateur via des fonctions de plus haut niveau (langage C), dans un esprit de prototypage rapide, d'implanter n'importe quelle application de traitement d'image sur une architecture multiprocesseurs homogène sans avoir connaissance de l'architecture matérielle. Par exemple, la fonction *farm()* permet de mettre en œuvre le squelette FARM (expliqué dans le chapitre 1). Une fonction appelée *synchro()* sert à la synchronisation entre deux processeurs.

Dans cette section, nous avons présenté un module, DMA-Routeur qui a été utilisé dans des travaux antérieurs. Nous avons discuté des améliorations que nous avons apporté à ce module en ajoutant deux topologies qui sont la grille et le tore, et en ré-usinant le code se qui nous a permit de réduire considérablement les ressources matérielles utilisées par celui-ci.

3.9 Gestion du flot vidéo

Dans cette section, nous décrivons deux modules qui ont été développés afin de gérer le flot vidéo dans le circuit (fig. 3.13).
Le système nécessite deux mémoires externes (avec swap) qui peuvent chacune contenir l'image globale. Chaque mémoire est connectée à un module *frame generator* à l'intérieur du circuit ASIC. L'un des deux modules est appelé *Input frame generator* (fig. 3.13), ce dernier est relié à des modules *frame grabber* implantés dans chaque nœud de l'architecture. Le nœud 0 est spécial car il gère le flot vidéo sortant via le module

Output frame generator. Mais avant d'aborder les modules, le protocole utilisé pour le signal vidéo est détaillé.

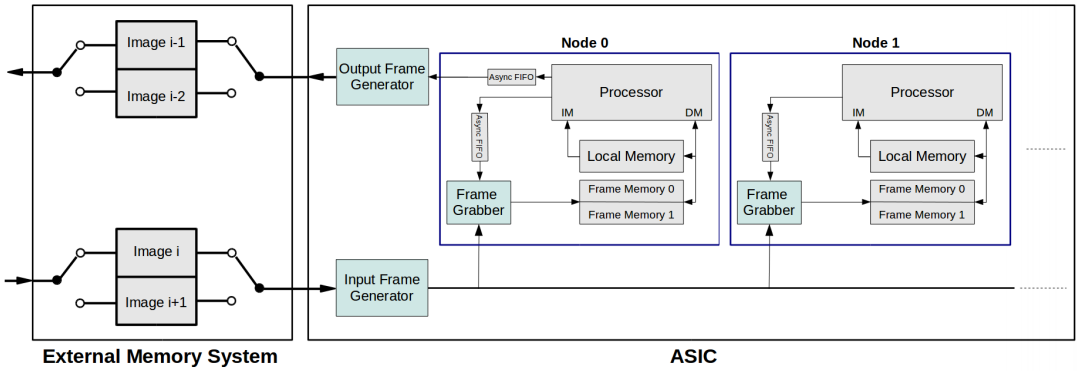


FIGURE 3.13 – Un système à mémoire externe pour sauvegarder l'image globale

3.9.1 Transport de l'information vidéo

Le protocole du signal vidéo utilisé permet d'accéder à n'importe quel pixel dans l'image. Comme le montre le chronogramme présenté sur la figure 3.14, le protocole est constitué de quatre signaux : un signal qui indique le début de la ligne (h_sync_i), un signal qui indique le début de l'image (v_sync_i), un signal qui contient la valeur du pixel (v_data_i) et vue que c'est un signal vidéo, un signal qui indique si c'est l'image est nouvelle (v_swap_i). Afin d'illustrée ces signaux, la figure 3.14 montre un chronogramme des quatre signaux pour deux image de taille 12X4 pixels qui se successive :

Dans la figure 3.14, le signal h_sync_i passe à 1 quatre fois indiquant ainsi le début de chaque ligne. le signal v_sync_i passe à 1 deux fois indiquant le début de deux images. Pour ce qui est du signal v_swap_i, celui-ci ne passe pas à 1 avec le premier v_sync_i au début du chronogramme, ce qui montre que cette image (l'image (i)) n'est pas nouvelle et qu'elle a été présente sur le bus au moins une fois avant. Lorsque ce signal passe à 1, il indique qu'une nouvelle image (l'image (i+1)) est disponible sur le bus. Pour ce qui de la valeur du pixel, le signal v_data_i est de taille 32 bits ce qui permet de faire circuler 4 pixels codés sur 8 bits à la fois, offrant plus de vitesse au flot vidéo lors de l'écriture de l'image dans la mémoire, car on écrit 4 pixels à la fois. Par exemple, pour une image 64x64 pixels = 4096 pixels , il faut 1024 coups d'horloge pour écriture l'image dans la mémoire. Donc à une fréquence de 200MHz, l'écriture de l'image dans la mémoire prend $5,12\mu s$.

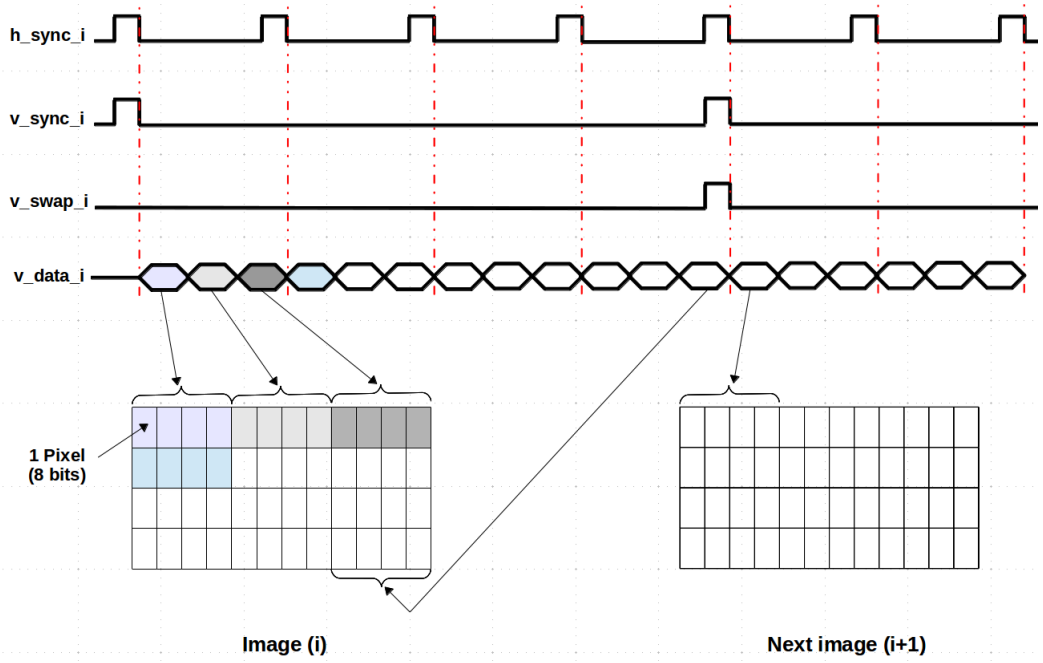


FIGURE 3.14 – Les composantes du bus vidéo.

3.9.2 Au niveaux nœud : *frame grabber*

Le *frame grabber* est un module implanté dans chaque nœud afin de lire le bus vidéo. Ce module est connecté au cœur du processeur par un lien FSL. Via ce lien le processeur envoie les informations concernant la portion d'image sélectionnée qui doit être écrite dans la mémoire vidéo individuelle de chaque nœud. La figure 3.16 montre la connexion entre le module et les quatre signaux du bus vidéo (en bas de la figure), la mémoire (à droite de la figure) et le processeur via un lien FSL (en haut de la figure).

Le module est constitué de cinq compteurs. Les deux compteurs *X offset* et *X length* permettent de sélectionner le coté x haut de la portion d'image. Les deux compteurs *Y offset* et *Y length* permettent de sélectionner le coté y droit de la portion d'image. Le compteurs *Pixel* permet de générer l'adresse pour l'écriture dans la mémoire. Le module *Registre File* est constitué de registres qui sauvegarde les informations reçues par le processeur (X offset, X length, Y offset et Y length). Le module *Control* reçoit les commandes du processeur, de la machine d'état *SWAP* (machine d'état qui mise on œuvre le swap), et permet aussi d'écrire dans les registres.

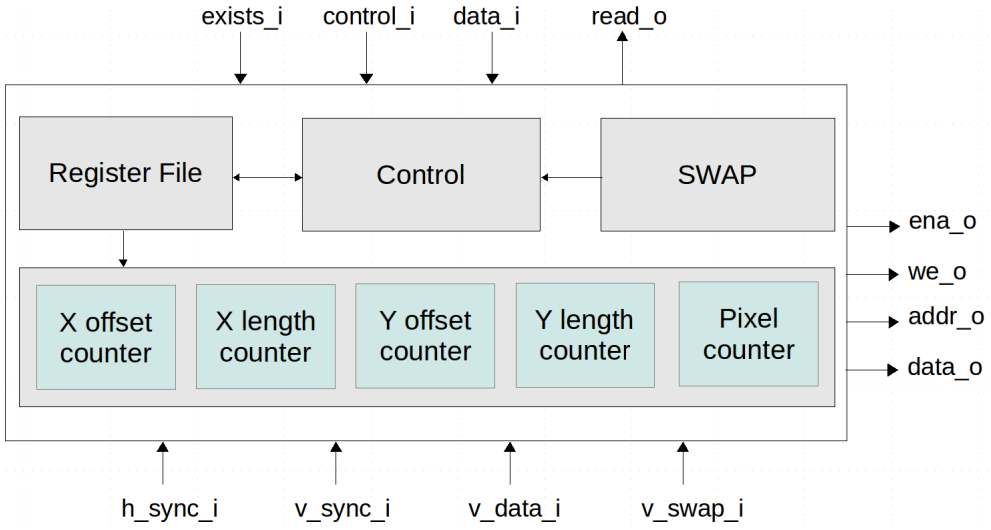


FIGURE 3.15 – Architecture du Frame grabber.

3.9.3 Au niveaux circuit : *frame generator*

Le *frame generator* est un module permettant le formatage du signal vidéo (*h_sync*, *v_sync*, *v_data* et *v_swap*) à partir du signal vidéo entrant. Le *frame generator* peut récupérer l'image de deux sources : la première en lisant la mémoire externe, la deuxième directement disponible via le lien FSL. Un point important concernant la communication via FSL est qu'elle permet deux modes de fonctionnement :

- Mode de configuration en utilisant les FSL de type contrôle. Grâce à ce mode, le module est configurable et peut supporter différentes tailles d'image.
- Mode opérationnel en utilisant les FSL de type data. Les données sont envoyées directement via des FSL qui sont mis en forme par le module pour finalement générer la donnée sur le bus vidéo.

Le *frame generator* est constitué de cinq compteurs. Les compteurs X et Y permettent de compter successivement les lignes et les colonnes. Le compteur *Pixel* permet de conter les pixels, le compteur *Frame* permet d'indiquer le nombre d'image, le compteur *IN-Pixel* permet de générer l'adresse pour la lecture de la mémoire. Le module contient aussi deux machines d'état, la machine d'état *SWAP* permet la gestion du swap et la machine d'état *Pack* permet de formater les pixels.

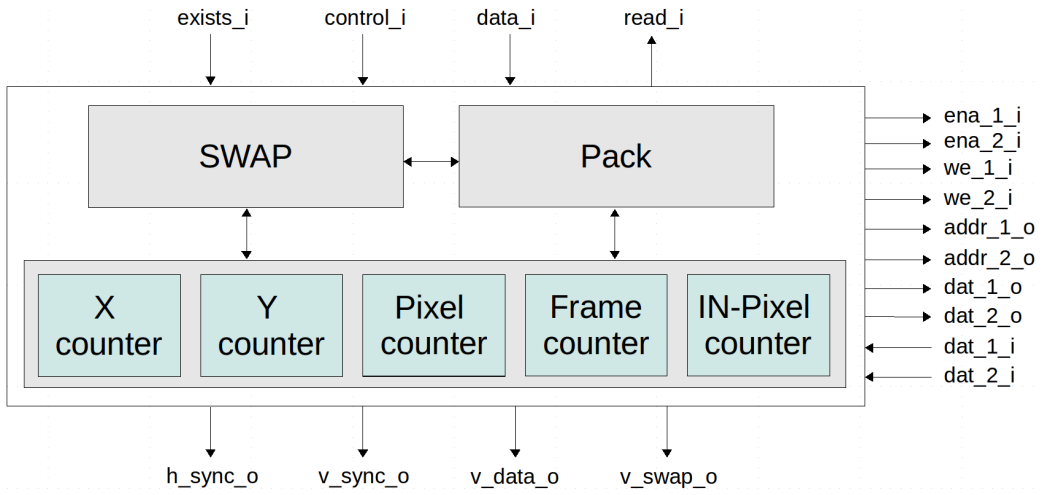


FIGURE 3.16 – Architecture du Frame generator.

3.9.4 Résultats d'implantation

Le tableau 3.15 présente l'utilisation des ressources matérielles des implantations des modules *frame grabber* et *frame generator* utilisant le Virtex-6.

	frame grabber	*frame generator*
Nombre de Flip-Flops	313 (0,10%)	131 (0,04%)
Nombre de LUTs	190 (0,13%)	139 (0,09%)
Nombre de Slices	174 (0,46%)	68 (0,18%)

TABLE 3.15 – Résultats d'implantation

3.9.5 Un système incluant une mémoire externe pour sauvegarder l'image globale

Ce module nécessite un développement matériel à l'extérieur du circuit (fig. 3.13). Il permet de gérer le flot vidéo à l'extérieur, c'est-à-dire entre le capteur d'image et le circuit (plus précisément le *frame generator*). Cette partie matérielle peut être développée et réalisée sur cible FPGA, mais globalement ce module doit être à la fois simple à mettre en œuvre sans pour autant prendre beaucoup de ressources matérielles. Ce module nécessite un système mémoire (fig. 3.13) qui peut contenir deux fois la taille de l'image entrante dans le circuit, afin de réaliser le swap entre l'image courante et l'image suivante sans avoir un temps d'attente. Le fait de mettre cette mémoire à l'ex-

térieur n'influe pas sur les performances du système, car il existe des mémoires qui peuvent attendre des fréquences de l'ordre de 250MHz. Par exemple pour une image de 1024x1024 pixels, et avec une lecture de 4 pixels par coup d'horloge, il faut 262144 coups d'horloge pour lire l'image entière. Ce qui fait environ 1ms (1048576ns).
Un autre argument qui nous a conduit à mettre cette mémoire à l'extérieur est que les mémoires intégrées (proposée par le fondeur) sont très volumineuses : elles occupent ainsi une surface importante sur le silicium ce qui induit une explosion du coût de fabrication (point discuté dans le chapitre suivant).

3.10 Conclusion

Dans ce chapitre, nous avons présenté la brique de base de notre MPSoC (le nœud). Nous avons discuté l'approche utilisée pour le choix du processeur et les améliorations que nous avons apportées à celui-ci en terme de communication (ajout d'instruction utilisateur de type FSL) et en terme de performance (ajout d'un système FPU). Nous avons expliqué l'approche mémoire choisie et la programmation de celle-ci via le JTAG. Nous avons vu aussi de la communication et plus précisément du DMA-Router et les améliorations qui lui ont été apportées (ajout de nouvelle topologies et ré-usinage). Nous avons présenté également l'approche utilisée pour la gestion du flot vidéo à l'intérieur et à l'extérieur du nœud.
Après avoir défini ce nœud de base (et complètement "open source") et l'avoir validé sur FPGA dans cette section, la phase de développement ASIC peut commencer. Le prochain chapitre est donc dédié à notre premier circuit de validation de ce nœud.

Chapitre 4

HNCP-I : conception du nœud de base en technologie ST 65nm CMOS

4.1 Introduction

Dans ce chapitre, nous présentons le premier circuit de type ASIC réalisé au sein de l'institut Pascal. La fabrication de ce circuit permet de valider le cœur du processeur (SecretBlaze) sur cible ASIC. Le cœur de ce processeur constitue la brique de base du futur MPSoC. L'objectif de ce premier RUN [1] est la preuve de concept et de faisabilité et de ce fait les contraintes de conception (tel que la fréquence) sont relâchées.

Dans un premier temps, nous allons présenter la technologie utilisée pour ce premier RUN ainsi que l'architecture implantée. Nous détaillons ensuite le flot de conception utilisé pour la fabrication du circuit avec les différentes étapes associées à la conception et vérification du circuit. Avant de conclure ce chapitre, nous présentons la carte de test développée permettant d'évaluer notre ASIC.

4.2 Choix de la technologie

Travailler directement avec un fondeur de circuits intégrés tel que STMicroelectronics [69], austriamicrosystem [70] ou encore TSMC [71] n'est pas possible pour un projet en petit volume tel que le nôtre. De ce fait, la technologie utilisée est fournie via le CMP (Circuit Multi Project) [72] qui est un service de fabrication de circuits intégrés, MCMs et microsystèmes, en prototypage et en petit volume, pour les universités, laboratoires de recherche, sociétés, en France et à l'étranger. Le CMP est un intermédiaire entre les concepteurs et les différents fondeurs. La figure 4.1 montre l'organisation et les délais des RUNs entre le client, le CMP et le fondeur.

1. Terme qui désigne le cycle de fabrication d'un circuit chez le fondeur

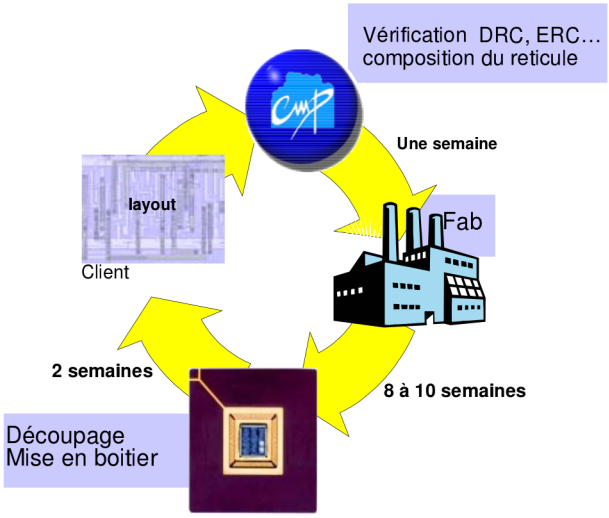


FIGURE 4.1 – Organisation des RUNs [72]

Le CMP collecte les dessins de masques de circuits de plusieurs de ses clients. Il les classe par technologie et par fonderie, puis les arrange de façon à ce qu'ils soient tous réalisés sur le même wafer (fig. 4.2). Un wafer est un disque de silicium d'AsGa ou d'un autre semi-conducteur sur lequel sont gravés les circuits. Le diamètre du wafer est exprimé en pouce. Il dépend essentiellement de la technologie utilisée pour réaliser le wafer. En technologie Silicium, les wafers font 8" et 12" alors qu'en AsGa, ils font 4" et 6". Une fois cette opération faite, l'ensemble wafer-masques est envoyé chez les fondeurs [73].

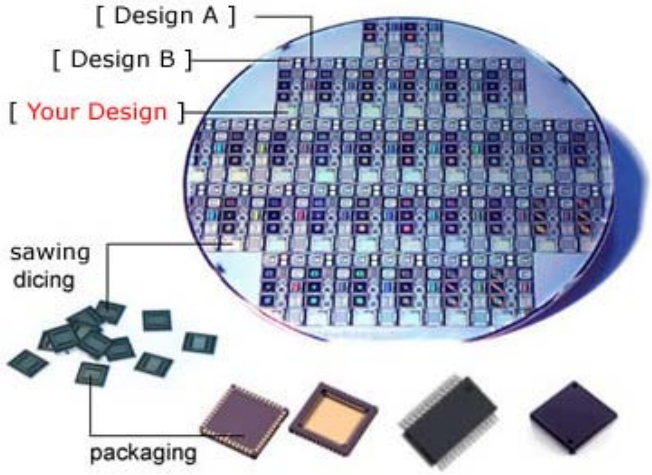


FIGURE 4.2 – Plusieurs circuits sur un même wafer[72]

Cette méthode permet de réduire le coût des circuits pour les organismes qui fabriques des circuits en faible volume. Il existe d'autres organismes offrant ce type de service tel que le CMC [74] au Canada, MOSIS [75] au USA, CIC au Taïwan, IDEC en Corée et VDEC [76] au Japon. Le choix du CMP était évident vu la situation géographique du CMP basé à Grenoble (France) : un contact et un support direct et rapide pour notre premier circuit était un critère important.

Lors du lancement du projet ASIC né 2012, nous avions le choix entre sept technologies disponibles via le CMP. Le tableau ci-dessous (tab.4.1) donne le coût du mm^2, la surface minimale et le nombre de RUN pour l'année 2012 pour les sept technologies.

Technologies	**Coût par mm^2 (€)**	**Surface minimale (mm^2)**	**RUNs par année**
AMS 350 nm	600	3	6
AMS 180 nm	1290	14	4
ST 130 nm	2200	1	4
ST 65 nm	7500	1	3
ST 40 nm	10000	1	1
ST 28 nm	15000	1	5
Tezzaron 3D 130 nm	1500	5	1

TABLE 4.1 – Coût, surface minimale et nombre de RUN donnée par le CMP pour l'année 2012

Pour ce qui est de la technologie Tezzaron 3D 130 nm, celle-ci n'a pas été retenue car elle ne correspond pas à nos architectures dans le cadre de cette thèse. Des estimations ont alors été effectuées sur les modules constituant le nœud présenté dans le chapitre (SecretBlaze, bus wishbone, mémoire, DMA-Router et divers les périphériques). Le tableau 4.2 présente une estimation de surface occupée par le cœur du Secretblaze [2] sur les quatre technologies AMS 350nm, AMS 180nm, ST 130nm et ST 65nm. La synthèse logique a été réalisée en utilisant RTL Compiler de chez Cadence. Ces évaluations ont été réalisées afin d'avoir un ordre de grandeur du coût du circuit et ont permis de retenir la technologie ST 65nm.

	AMS 350nm	AMS 180nm	ST 130nm	ST 65nm
Nombre de cellules	12549	10695	11497	10162
Surface (mm^2)	1,436	0,292	0,159	0,047
Coût (€)	861,6	376,7	349,8	352,5

TABLE 4.2 – Évaluation du cœur du SecretBlaze sur les quatre technologies

2. Surface après l'étape de synthèse logique (sans cellules mémoires et sans cellules IOs).

La technologie ST 65 nm est très utilisée, stable et nous permet d'avoir un support efficace sur une technologie éprouvée. La figure 4.3 montre le nombre de design-kits (données de processus technologiques et la bibliothèque de composants fournie par le fondeur) de chez STMicroelectronics distribué par le CMP depuis 2006 jusqu'à 2012. Pour la technologie 28 nm, nous observons (fig. 4.3) que la distribution du design-kits est récente. Même si la distribution a débuté en 2011, les RUNs on commencé en 2012. Pour la technologie 40 nm, le calendrier du CMP [72] montre 4 RUNs en 2011, puis 1 RUNs en 2012 et enfin 2 RUNs étaient prévu en 2013 mais un seul est retenu (celui du mois de janvier). Au final (en juin 2012), le choix d'une technologie de pointe mais éprouvée a été préférée aux technologies moins stable (ST 40nm) ou en tout début de cycle de vie (ST 28nm).

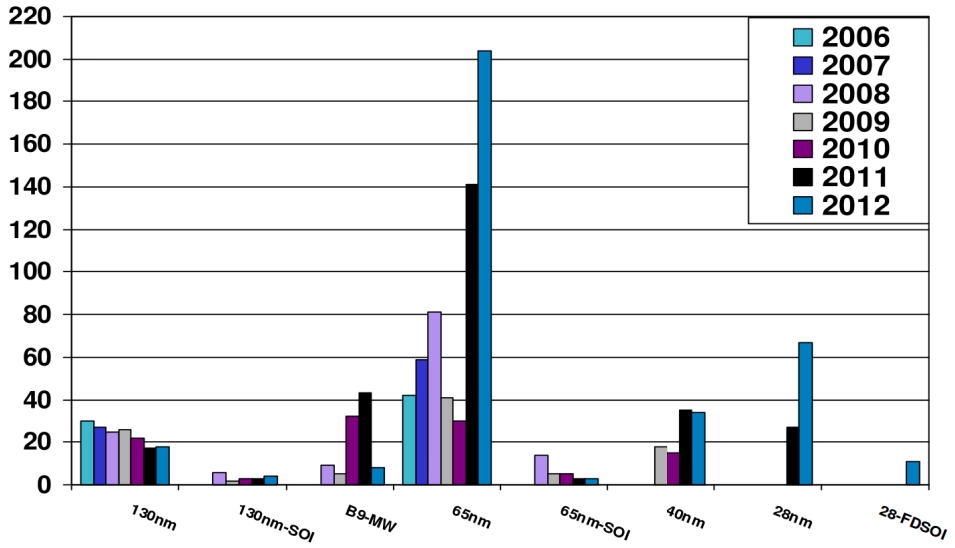


FIGURE 4.3 – Distribution du design-kits STMicroelectronics par technologie [72]

Un autre point important pour justifier le choix de la technologie doit être évoqué : les projets en ST CMOS 65 nm allant jusqu'à l'étape de fabrication pour l'année 2012 étaient de 90 (pour 306 centres ayant reçu le design-kit), soit un taux de réalisation de 29,41 %. Même si les chiffres de la figure 4.3 montre une distribution importante des design-kits, la fabrication de circuits n'est pas aussi importante, ce qui montre les difficultés rencontrées par les concepteurs, soit par rapport au coût de fabrication qui reste toujours élevé même via des organismes tel que le CMP, soit par rapport à la conception et au savoir faire des ASIC, qui devient de plus en plus difficile avec les technologies fortement intégrées. A titre d'information, pour la ST CMOS 40 nm, le taux de réalisation de circuit est de 6,02 % en 2012 (5 runs pour 83 centres ayant reçus le design-kit). Pour la ST CMOS 28 nm, 103 centres on reçu le design-kits quand le nombre de circuits fabriquées était de 22 ce qui fait 21,35% de réalisation.

Le process retenu -CMOS 65 nm de chez STMicroelectronics- présente les caractéristiques suivantes :

- longueur du poly 65 nm,
- double ou triple transistors MOS Vt,
- double ou triple oxyde de grille,
- process dédiés à la haute performance ou à la faible puissance,
- cuivre double damascène pour interconnexion,
- faible-k (k = 2,9) diélectrique,
- 6 ou 7 couches de métal,
- pas de métallisation de 0,20 micron,
- capacités analogiques/RF,
- 800 K portes/mm^2,
- support diverses alimentations : 2,5 V, 1,8 V, 1,2 V, 1V,
- mémoire embarquée (simple port RAM /ROM /double port RAM).

En terme de cellules standards, STMicroelectronics propose des librairies de cellules appelés CORE à usage général (General purpose core libraries). Le fondeur propose aussi CORX qui sont des librairies de cellules complémentaires, CLOCK qui sont des librairies de cellules utilisables surtout pour l'arbre d'horloge et les FILLER appelés PR (pour le placement-routage). Des librairies DP (Datapath leaf) et HD(High Density) sont proposés sur commande. Pour ce qui est des librairies de cellules d'entrées/sorties (IO), diverses tension (1,8V, 2,5V et 3,3V) avec diverses dimensions (80μ, 65μ, 60μ, 50μ 40μ et 30μ) sont disponibles pour du numérique et de l'analogique. Il y a aussi des convertisseurs de niveau et des cellules de compensation pour les systèmes à grande surface. Les mémoires proposées sont de type SPRAM, DPRAM ou encore ROM, disponibles sur commande, mais sans frais supplémentaires. Les mémoires ne sont pas considérées comme des cellules standards, de ce fait, les avoir gratuitement est un plus. D'autres fonderies font payer les mémoires en plus de la surface occupée par celle-ci sur le silicium. Des blocs IPs tels que des PLL, DLL, broches LVDS sont disponibles également sur commande et payantes.

4.3 Étude et choix de l'architecture du RUN

L'objectif de ce premier run est de fabriquer une brique de base (avec une contrainte financière forte) pour valider l'aspect communication via les instructions FSL ajoutées au processeur. La validation des communications via instructions FSL permet de valider la communication et le contrôle à l'intérieur et aussi à l'extérieur du nœud (communication cœur avec cœur, cœur avec DMA-Routeur, cœur avec Frame graber/generator, cœur avec l'unité de contrôle FPU). Ce circuit doit permettre également de valider le fonctionnement du JTAG dans les deux modes : programmation et debug.

Après une analyse de plusieurs configurations d'architectures, nous avons constaté qu'il y a deux paramètres principaux impactant la surface du circuit. Le premier pa-

ramètre est la mémoire, car un bloc mémoire de taille 256 x 8 bits occupe une surface de 10116 (μm^2). De ce fait, nous avons choisi de prendre une mémoire de taille 4 Ko suffisante pour implanter divers programmes de test du processeur. Le deuxième paramètre est le nombre des broches d'entrée/sortie : effectivement même si la partie active du circuit est de faible dimension, la surface totale du circuit est contrainte (et augmentée) par le placement en anneau des IOs (pad ring). Ainsi, quatre architectures ont été évaluées en terme de surface afin d'estimer le coût du circuit :

- La première architecture proposée (fig. 4.4) est constituée de deux cœurs SecretBlaze connectés via deux ports FSL. Un des cœurs est connecté à l'extérieur via des port FSL.

- La deuxième architecture, toujours à base de deux cœurs, est identique à la seule différence que tous les ports FSL sont re-bouclés sur eux mêmes (auto test). Par conséquent, le nombre d'entrées/sorties du circuit diminue (44 broches d'entrées/-sorties de moins que la précédente).

- La troisième architecture proposée est constituée d'un cœur SecretBlaze. Le cœur possède quatre ports FSL dont deux sont re-bouclés sur eux-même et les deux autres sont connectés à l'extérieur.

- La quatrième architecture (fig. 4.5) est la même que la précédente à la seule différence que tous les ports FSL sont re-bouclés sur eux-mêmes.(44 broches d'entrées/sorties de moins). L'architecture contient $4Ko$ de mémoire programmée via le JTAG. La version du JTAG mise en œuvre est adaptée du standard IEEE 1149.1 JTAG. Quatre périphériques (UART, TIMER, GPIO and INTC) sont connectés au bus Wishbone. Le module "clock" permet de générer deux horloges, une pour le CPU (cadre rouge sur la figure) et l'autre pour le bus wishbone et les périphériques. La fréquence de cette horloge est égale à la moitié de la fréquence d'horloge dédiée à la CPU. Le module "Reset" permet de configurer le circuit (mode fonctionnel, mode programmation, mode debug).

Les quatre architectures présentés précédemment ont pour objectif de valider une brique de base d'une future architecture multiprocesseurs basée sur les concepts de la méthode HNCP. Le tableau ci-dessous (tab. 4.3) récapitule les paramètres de surface (voir le chapitre 1, sous-section : Les ASICs) des quatre architectures et le coût total du circuit pour 25 exemplaires, dont 20 sans boîtier. Le coût du millimètre carré pour la technologie ST 65nm est égal à 7500 €. Ce coût est le même pour tout circuit dont la surface est inférieure à $5mm^2$, ce qui est le cas pour les quatre architectures. L'architecture choisie est celle composée d'un cœur sans port FSL (extérieur : 4ème architecture). Comme nous pouvons le voir sur le tableau 4.3, cette architecture offre le coût le plus bas parmi les quatre proposées et permet de remplir les objectifs du premier RUN (valider une brique de base construite autour d'un processeur).

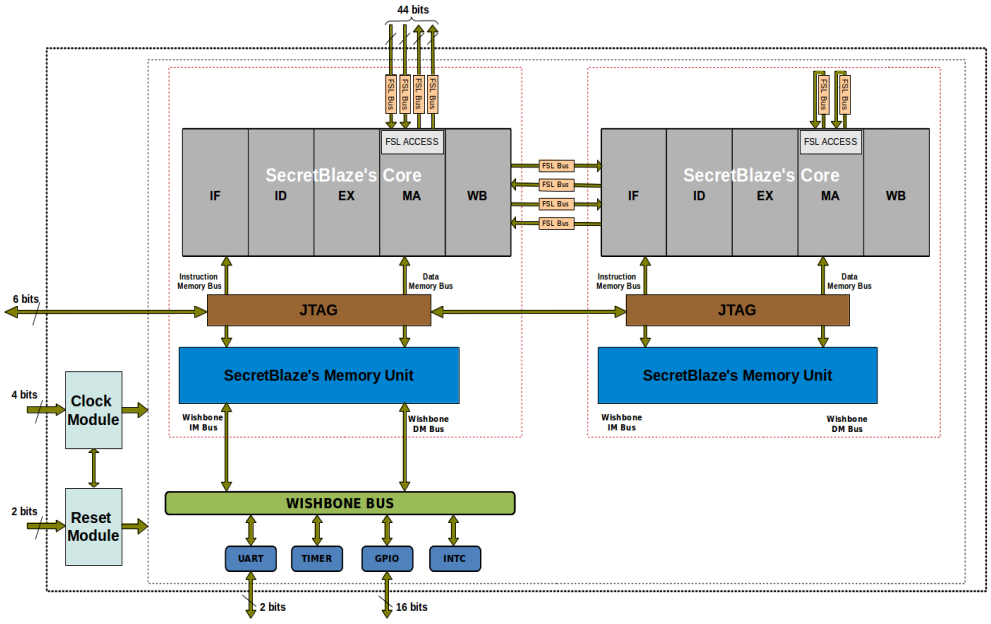


FIGURE 4.4 – Architecture à base de deux cœurs avec ports FSL

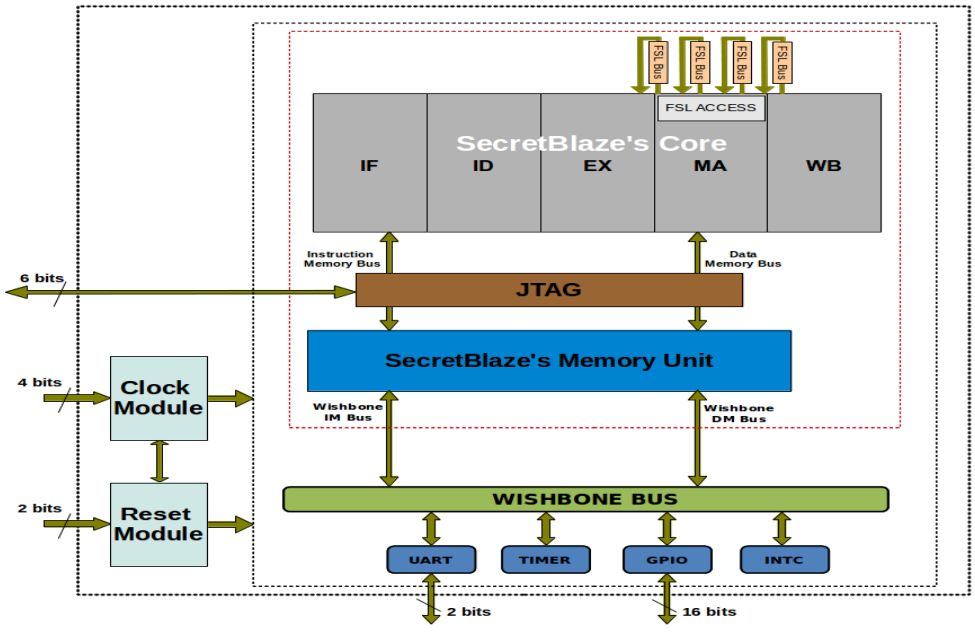


FIGURE 4.5 – Architecture à base d'un seul cœur sans port FSL

	Un cœur avec port FSL	**Un cœur sans port FSL**	**Deux cœurs avec port FSL**	**Deux cœurs sans port FSL**
Nombre de cœurs	1	1	2	2
Nombre d'IOs	91	46	91	46
Largeur de la surface active (μm)	799	709	1114	1109
Hauteur de la surface active (μm)	844	744	1104	1099
Largeur de la surface active + IO(μm)	1215	1125	1530	1525
Hauteur de la surface active + IO (μm)	1260	1160	1520	1515
Largeur du seal ring (μm)	120	120	120	120
Largeur des IOs (μm)	108	108	108	108
Distance entre surface active et IO (μm)	100	100	100	100
Coût d'un 1 mm^2(€)	7500	7500	7500	7500
Surface active (μm^2)	674356	527496	1229856	1218791
Surface active + IO (μm^2)	1530900	1305000	2325600	2310375
Surface active + IO + seal ring (μm^2)	2182500	1911000	3115200	3097575
Coût du RUN (€)	**16 368,75 €**	**14 332,50 €**	**23 364,00 €**	**23 231,81 €**

TABLE 4.3 – Estimation de surface et de coût d'un ASIC en technologie ST 65nm pour quatre architectures envisagées

4.4 Méthodologie utilisée pour le test du circuit

Une méthodologie de test a été mise en place en vue d'assurer le bon fonctionnement du futur circuit réalisé. La stratégie de test de notre circuit (conception en vue du test) est implantée en mettant en place un système de débogage présenté sur la figure 4.6. Ce module additionnel a pour but d'améliorer la contrôlabilité et l'observabilité des signaux du circuit et donc d'améliorer la testabilité globale du circuit. Le système est basé sur les principes classiques du JTAG. Nous pouvons suivre l'exécution de trois instructions : *load*, *store* et *FSL* entre tous les étages du pipeline. Le principe est le suivant : à chaque coup d'horloge on peut contrôler ou observer jusqu'à 461 signaux (c'est à dire imposer ou lire leur contenu) permettant ainsi de tracer un chronogramme de l'évolution de ces instructions dans le pipeline et donc identifier l'éventuel étage posant problème.

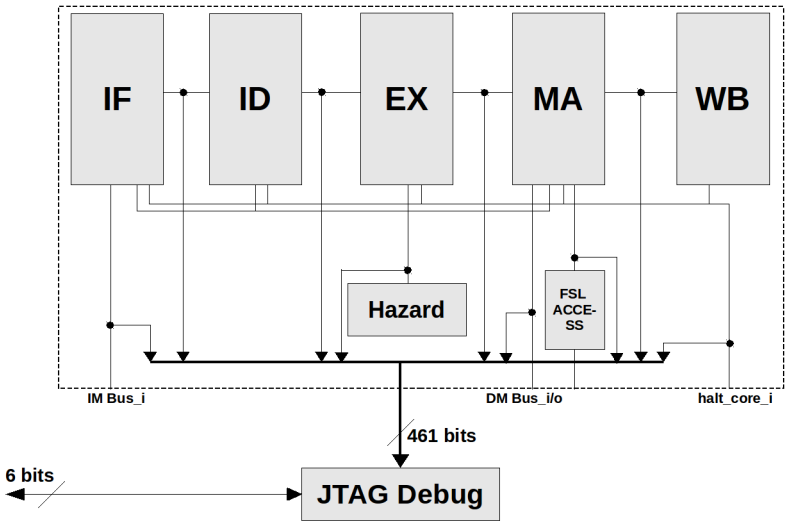


FIGURE 4.6 – Architecture permettant le debug du circuit via le JTAG

4.5 Le prototypage sur FPGA

L'utilisation d'un FPGA pour le prototypage d'ASIC est maintenant devenu une pratique courante. L'idée principale est de créer un contexte aussi proche que possible que celui de l'ASIC fabriqué. Plusieurs simulations et implantations ont été effectuées sur la cible FPGA avec divers programmes afin de valider tous les blocs de l'architecture (programme qui utilise l'URAT, programme qui utilise le Timer,...). L'architecture proposée a été implantée sur une carte de développement de chez Xilinx. Elle a été synthétisée et placée-routée avec la version 14.3 de XST (Xilinx Synthesis Technology), utilisant un FPGA Spartan-6 XC6SLX45 (SpeedGrade : -3) de technologie 45nm. Les ressources évaluées après placement-routage sont 3255 Flip-Flops, 4615 LUTs et 16

BRAMs. La fréquence d'entrée de l'architecture est de 100MHz. La fréquence maximale donnée par le synthétiseur est de $126,610$MHz. En appliquant une fréquence de 100MHz sur la CPU et 50MHz sur le bus Wishbone et les périphériques (GPIO, UART, INTC et Timer). La consommation statique de cette architecture est estimée sur le même FPGA (Spartan-6 XC6SLX45 (SpeedGrade : -3)) à $0,038W$ et la consommation totale (statique + dynamique) est de $0,126W$. Le système de prototypage est présenté en **annexe B.1**.

4.6 Conception de l'ASIC

Dans cette section, nous présentons le flot et les outils de conception utilisés dans ce premier RUN. Nous allons introduire et définir l'approche utilisée pour la migration du code RTL pour la cible ASIC. Nous illustrons par la suite les résultats de synthèse logique, placement-routage et simulations. Nous finissons par l'étape de vérification.

4.6.1 Flot et outils de conception du premier RUN

Le flot de conception est basé sur des outils de CAO comme indiqué sur la figure 4.7. Les outils de CAO utilisés doivent être compatibles avec le design-kit choisi fourni via le CMP ce qui contraint les choix d'outils et de version. Le flot de conception commence par la spécification de l'architecture, le code RTL déjà validé par le prototypage FPGA (avec des modifications du code) est synthétisé grâce à l'outil de synthèse logique Design Compiler de chez Synopsys (on a utilisé aussi RTL Compiler de chez Cadence). Après plusieurs modifications du code RTL (voir la section Migration), la simulation après synthèse est fonctionnelle (l'outil de simulation utilisé est NCSim de Cadence). Ensuite, le placement-routage de l'architecture est réalisé avec l'outil EDI (Encounter Digital Implementation) de Cadence afin d'obtenir un fichier au format GDSII, image des dessins de masque du circuit. L'outil de Cadence Virtuoso permet de faire des retouches ou ajustements au niveau du layout comme fixer la taille des cellules d'entrées/sorties (IOs). Par exemple, il nous a fallu changer la taille des cellules IO de $40\mu m$ à $50\mu m$ pour faire face aux contraintes de mise en boîtier (packaging) imposées par le CMP. L'étape suivante est l'étape de vérification. Les vérifications DRC (Design Rules Check) et LVS (Layout Versus Schematic) ont été réalisées avec Calibre, car même si l'outil Virtuoso permet de faire ce type de vérification, celui-ci n'est pas adapté avec le design-kit ST CMOS 65nm. Enfin, le GDSII est soumis au CMP pour une deuxième passe de vérification.

4.6.2 Approche utilisée pour la migration du code RTL pour la cible ASIC

Lors de la simulation après synthèse, celle-ci ne fonctionnant pas, nous déduisons que la synthèse était incorrecte. Pour résoudre ces problèmes et du fait de la complexité de l'architecture, nous avons utilisé une approche de vérification modulaire : l'architecture a été divisée en plusieurs modules (ID, IF, EX, UART, bus Wishbone ...), chaque

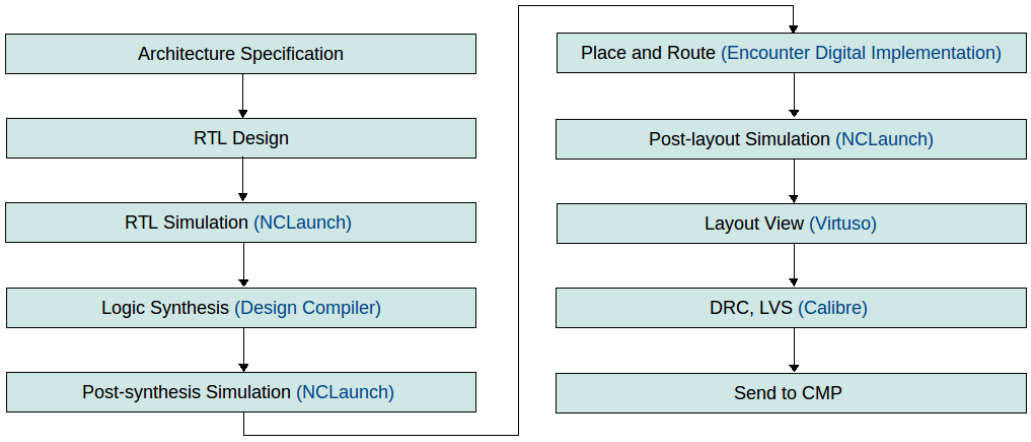


FIGURE 4.7 – Flot de conception du premier RUN

module a été synthétisé seul afin de générer sa propre netlist. Pour valider un module donné, nous simulons toute l'architecture au niveau RTL, sauf pour le module en cours de validation qui est remplacé par la netlist synthétisée (le code RTL peut être vu comme un testbench pour la netlist en question). Avec cette méthodologie et après plusieurs modifications du code RTL, la simulation fonctionne correctement. La figure suivante (fig. 4.8) illustre une validation de l'étage IF.

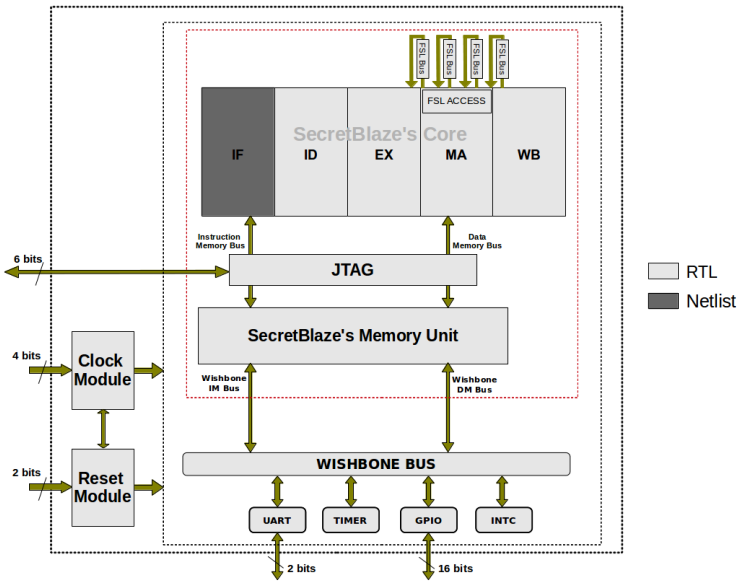


FIGURE 4.8 – Architecture permettant la validation de l'étage IF (Instruction Fetch)

4.6.3 La synthèse logique

Le tableau 4.4 présente les résultats de synthèse logique en terme de surface et de nombre de cellules pour les 11 blocs décrits dans l'architecture choisie. La surface totale de cette architecture est de $0,236mm^2$ avant ajout des cellules d'entrées/sorties et avant placement-routage.

Modules	**Nombre de cellules**	**Surface (μm^2)**	**Consommation statique (μW)**
SecretBlaze's Memory Unit	624 (4,24%)	164648 (69,81%)	227,55 (90,48%)
SecretBlaze's Core	10938 (74,29%)	52762 (22,37%)	16,71 (6,64%)
JTAG	1753 (11,91%)	11134 (4,72%)	4,65 (1,85%)
FSL Bus	45 (0,30%)	362 (0,15%)	0,15 (0,06%)
WishBone Bus	112 (0,76%)	360 (0,15%)	0,09 (0,04%)
UART	273 (1,85%)	1269 (0,54%)	0,44 (0,17%)
TIMER	571 (3,88%)	2968 (1,26%)	1,01 (0,40%)
GPIO	63 (0,43%)	332 (0,14%)	0,11 (0,05%)
INTC	183 (1,24%)	794 (0,34%)	0,27 (0,11%)
Clock,Reset,...	162 (1,10%)	1231 (0,52%)	0,51 (0,20%)
Top level	**14724 (100%)**	**235860 (100%)**	**251,49 (100%)**

TABLE 4.4 – Résultat de la synthèse logique

Pour ce qui est de la consommation dynamique, le synthétiseur (Design Compiler) donne un résultat de consommation interne des cellules (Cell Internal Power) de $43,03mW$ à une fréquence de 200MHz.

4.6.4 Le placement-routage

L'unité de mémoire couvre la majeure partie de la zone active du circuit comme nous pouvons le constater en position nord-ouest de la zone active (voir les 16 blocs de la figure 4.9). La taille de la mémoire choisie est équivalent à 16 blocs de $256x8$ bits. La surface totale du circuit est $0,906mm^2$ (deltaX = $0,952mm$ et deltaY = $0,952mm$). L'analyse de la consommation d'énergie donne une consommation statique de $0,31mW$ et une puissance totale de près de $70mW$ ($CV^2F = 0,07W$) (puissance statique et dynamique) pour l'ensemble du circuit. Ce résultat est très faible, notamment par rapport à une implantation similaire sur une cible FPGA de technologie comparable (voir les $126mW$ lors de l'étape de prototypage sur cible FPGA). Le circuit peut fonctionner à une fréquence d'horloge de 200MHz. Nous pouvons remarquer que la surface totale est inférieure à celle estimée précédemment (on est passé de $1,305mm^2$ à $0,906mm^2$), ceci est dû à l'optimisation effectuée lors de l'étape de placement-routage mais également en passant de 46 IOs à 38 IOs en ne gardant que 8 IOs pour l'alimentation au lieu de 16. Le coût du circuit est estimé à 10 656 € sans encapsulation (packaging).

La puissance dynamique a été calculée à l'aide de l'expression CV^2F en considérant le cas critique (pire cas) d'un taux d'activité égal à un (C : capacitance $(0,30nF)$, V : tension d'alimentation $(1.1V)$ et F : fréquence d'entrée (200MHz))

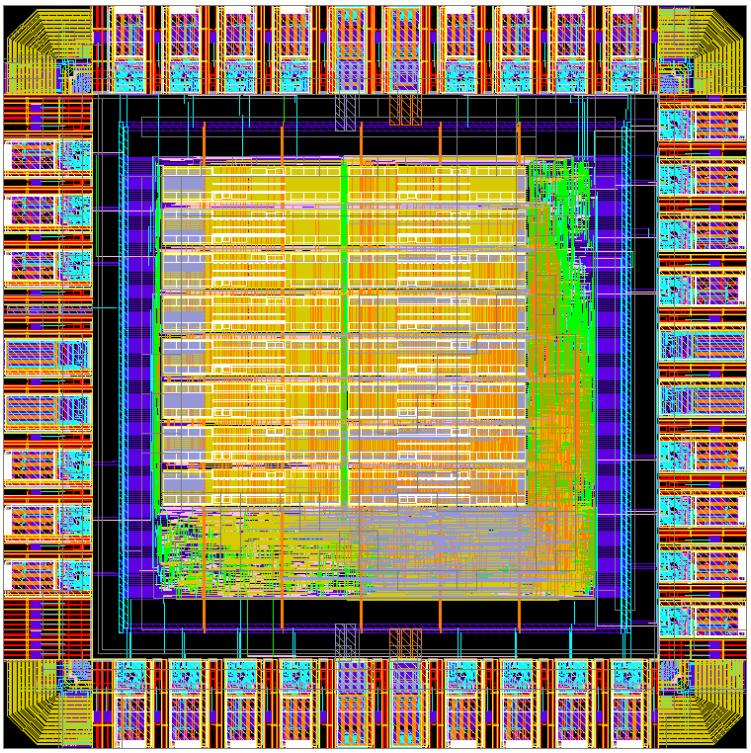


FIGURE 4.9 – Layout du circuit vue sous virtuoto

Le tableau 6.4 montre la consommation statique du circuit en fonction des librairies utilisées. D'une part, les mémoires (DPHS) consomment 73,34% alors qu'elles ne représentent que 0,08% de nombre de cellules (un bloc mémoire de $256x8$ bits présente une surface de $10116\mu m^2$ et une consommation statique de $0,014mW$). D'autre part, les cellules standard (CORE65LPSVT et CLOCK65LPSVT) consomment 10,57% et représente 98,22% du nombre de cellules.

Librairies	**Nombre de cellule**	**Consommation Statique** (mW)
DPHS	16 (0,08%)	0,23 (73,34%)
CORE65LPSVT	18640 (98,22%)	0,03 (10,57%)
CLOCK65LPSVT	152 (0,80%)	0,0002 (0,08%)
IO65LP_SF_BASIC_50A	140 (0,74%)	0,04 (12,34%)
IO65LPHVT_SF_1V8_50A	30 (0,16%)	0,01 (3,67%)

TABLE 4.5 – Résultats de consommation statique après placement-routage par librairies

4.6.5 Les simulations

Trois simulations on été classiquement réalisées tout au long du flot de conception ASIC : une simulation au niveau RTL, une simulation après synthèse (post-synthesis) et enfin une simulation après placement-routage (post-layout). Toutes ces simulation étaient fonctionnelles avec les mêmes programmes utilisés lors du prototypage (afin de valider tous les blocs de l'architecture). Le chronogramme montrant le résultat de simulation d'une exécution de programme classique appelé "hello world" est présenté en **annexe B.2**.

4.6.6 Vérification physique et envoi du circuit

Le circuit a également été vérifié en utilisant les outils (fig. 4.7) de DRC et de LVS de la suite Calibre de chez Mentor Graphics. Le résultat de cette vérification étant positif (0 erreur), le circuit était prêt à être envoyé au CMP. Le numéro de référence de notre RUN est S65C14_1. La date limite de soumission était fixé au 17 mars 2014. La date de livraison prévue du circuit était le 15 août 2014. Finalement, celui-ci a été livré le 10 septembre 2014.

4.6.7 Mise en boîtier du circuit

La méthode utilisée pour établir les connexions entre les IOs du layout et les broches du boîtier s'appelle le câblage par fil (Wire bonding). La figure 4.10 montre le câblage de fil appliqué à notre circuit avec un boîtier de type PGA68 (Pin Grid Array soit une matrice de broches, au nombre de 68 dans notre cas).

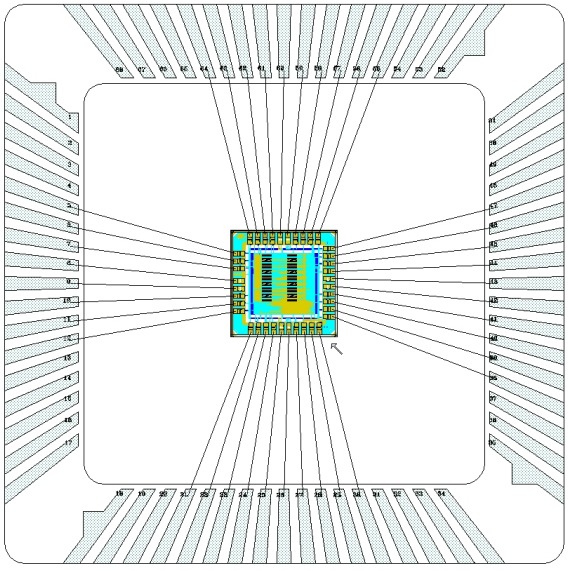


FIGURE 4.10 – Câblage de fil donné par le CMP

La figure 4.11 montre le circuit dans son boiter, celui-ci est encapsulé dans un PGA68 (le plus petit disponible via le CMP dans cette catégorie) :

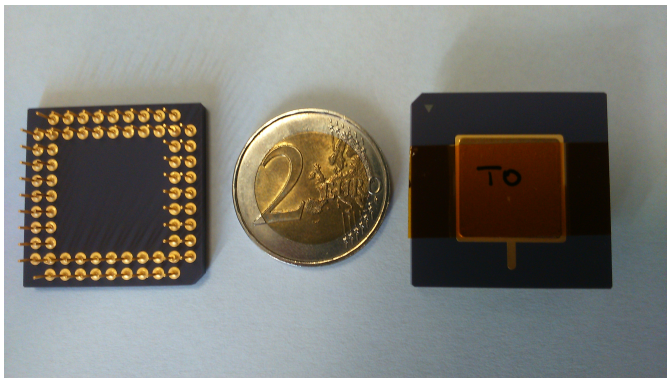


FIGURE 4.11 – Le premier circuit fabriqué à l'institut Pascal

4.7 Carte de test

La carte de test a été mise en œuvre avant la réception du circuit dans un esprit d'anticipation. Afin d'avoir une flexibilité maximale lors du test du circuit, nous avons choisi de connecter tous les ports d'entrées/sortie (30 broches) à un FPGA. Ce qui nous permettra de bien contrôler les signaux d'entrées/sorties du circuit ASIC. La figure 4.12 montre les composants de la carte de test du premier RUN (la vue de dessous de la carte n'est pas présentée, car elle contient que des capacités de découplage).
Un support ZIF (Zero Insertion Force) **(1)** adapté au circuit fabriqué (PGA68) permet de placer et retirer le circuit sur la carte de test sans risque de détérioration. Comme mentionné précédemment, le FPGA **(2)** (Cyclone 3 EP3C16 de chez Altera fabriqué avec la même technologie (65nm)) permet de mettre en forme les signaux et de mieux les contrôler en appliquant des contraintes , spécialement de timing, lors du codage de l'architecture de test sur le FPGA. Un module USB Cypress a été ajouté à la carte permettant le transfert de données entre le PC et le FPGA, en mode bidirectionnel **(3b)**. Cette transmission est réalisée via un Micro-USB type B **(3a)**. Huit LEDs vertes **(4)** servant à visualiser les GPO (General-purpose output) et huit Switchs **(5)** servant à tester les GPI (General-purpose input) du circuit ASIC ont été ajoutés. Un Reset permet la réinitialisation du FPGA. Ont également été ajoutés : un dispositif RS232 pour la liaison série du circuit ASIC (UART) **(7b)**, un connecteur RS232 **(7a)**, deux rangées de connecteurs (2x13) **(8)** qui accueil le module FT2232H utilisé lors du prototypage (fig. B.1) et deux connecteurs de type T821. L'un **(9)** est utilisé dans le cas d'absence du JTAG via **(8)**. Le deuxième connecteur **(10)** sert à connecter le USB-Blaster, ce dernier permettant de programmer le FPGA. Un oscillateur **(11)** indispensable pour la génération de l'horloge de référence a été rajoutée. Un autre module indispensable pour le fonctionnement du FPGA est le dispositif de configuration série (EPCS) **(12)**.

Afin de faciliter le debug, nous avons rajouté un connecteur analyseur de type AMP Mictor 34 voies femelles coudées **(13)** pour visualiser les signaux qui transitent entre le FPGA et le circuit ASIC. Toujours à des fins de test, des points de tests **(14)** des signaux à visualiser sont aussi mis en place. Un cavalier **(16)** pour la sélection de l'horloge à analyser (une à la fois) a été adjoint. Concernant l'alimentation de la carte, cinq régulateurs **(15c)** fournissent les tensions 5V, 3,3V, 2,5V, 1,8V et 1,2V. Un switch pour la mise sous tension de la carte avec une LED rouge **(15b)** et un connecteur pour câble d'alimentation 5V continu **(15a)** sont également rajoutés.

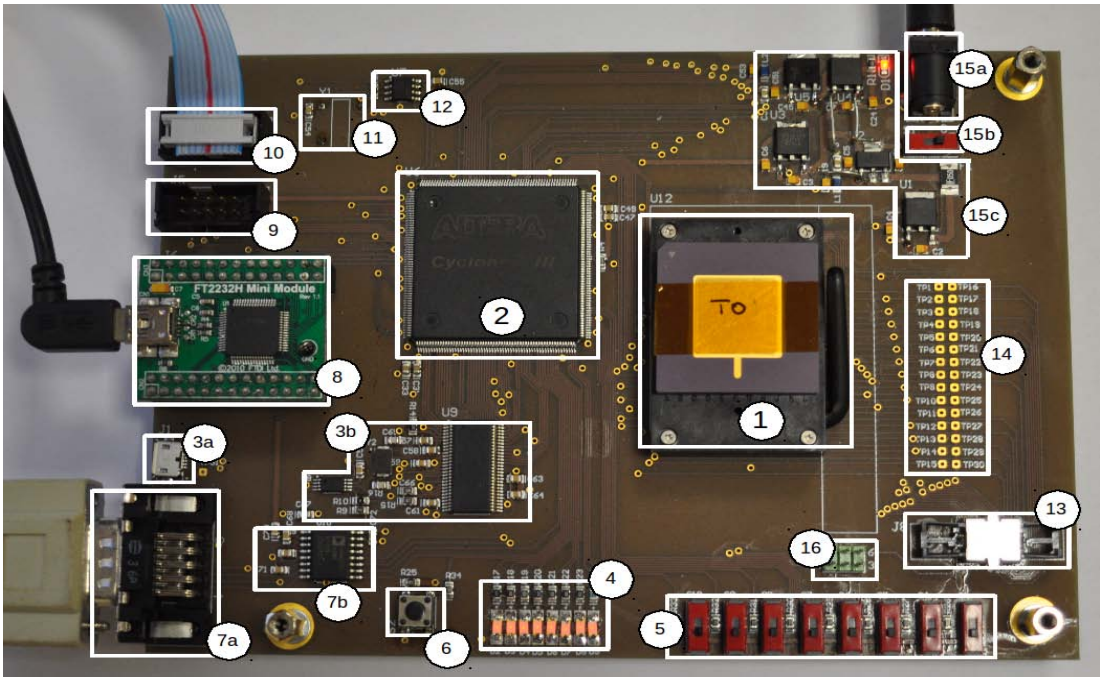


FIGURE 4.12 – Carte de test du circuit

4.8 Test du circuit

Dans cette section, nous parlons des tests effectués afin de valider le fonctionnement du circuit.
Dans un premier temps, nous avons validé la carte de test. Le premier test effectué est un simple test visuel à l'aide de la lunette binoculaire industrielle. Suite à cette première étape, des ajustements et corrections (re-soudage, changement de composants, etc ...) ont été apportés afin de rendre la carte opérationnelle.

Dans un second temps, la configuration du composant EPCS du FPGA est réalisée afin que ce dernier puisse être programmé. Un premier programme consistant à

connecter directement les 8 switchs aux 8 LEDs a permis de valider ces deux composants (switchs et LEDS). Puis, une implantation d'un compteur (validation du quartz et des LEDs) et un test de l'UART (connexion directe de Tx à Rx) ont été effectués. Enfin un test du JTAG par l'envoi via le module FT3222H (connecté sur les deux rangées de connecteur **(8)**) depuis le PC a été réalisé en observant à l'oscilloscope les signaux JTAG capturés entre le FPGA et le ZIF.

Dans un troisième temps, un simple programme "chenillard" a permis de tester l'ASIC en programmant directement la mémoire locale du processeur. Les étape effectuées sont les suivantes :

- Programmation du FPGA. L'architecture implantée sur le FPGA est présentée sur la figure 4.13. Elle est constituée d'une PLL qui génère une fréquence de 200MHz pour le cœur et la mémoire, et une fréquence de 100MHz pour le bus Wishbone et les périphériques. En plus de la PLL, des interconnexions permettent de relier l'ASIC aux périphériques de la carte de test.
- Mise en mode programmation du circuit (via le sw 4).
- Programmation de l'ASIC (via le FT2232H).
- Revenir en mode fonctionnel (via le sw 4).
- Effectuer un reset du processeur (via le sw 1).

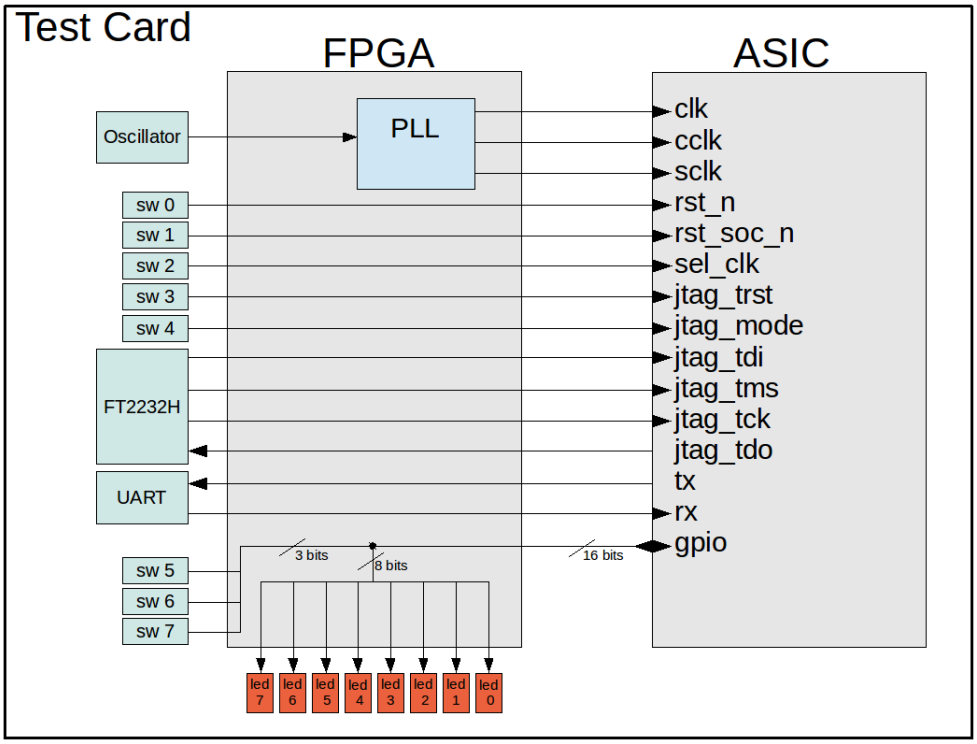


FIGURE 4.13 – Schéma du test du circuit

Ce test a montré que **le circuit est fonctionnel** ce qui implique la validation de la majorité des différents modules de l'ASIC. Les composants validés par ce programme sont les suivants :

- Le module de programmation JTAG (le JTAG debug n'est pas encore testé).
- La mémoire du circuit (tests en écriture et en lecture via le JTAG).
- Le cœur du processeur (instructions correctement exécutées).
- Le module "clock" génère bien les deux horloges nécessaires au fonctionnement du circuit (voir section 1.3).
- Le module "Reset" (passage du mode programmation au mode fonctionnel).
- Le bus Wishbone (accès au périphérique validé).
- Le périphérique GPIO.

4.9 Conclusion

Dans ce chapitre, nous avons présenté le premier circuit réalisé à l'Institut Pascal. En effet, nous avons proposé la brique de base d'un futur multiprocesseur. Grâce à ce projet nous possédons une solide base pour la conception d'ASIC numérique au laboratoire. Ce travail a aussi un intérêt dans le secteur académique, où on a pu mettre en place deux publications dans le domaine pédagogique. La première communication publiée dans CETSIS 2013 propose une formation destinée aux cycles d'ingénieurs et de masters. Elle aborde le thème de la conception des systèmes électroniques numériques. La deuxième communication publiée dans EWME 2014 propose un enseignement des méthodologies de pointe dans le domaine de la conception de circuit ASIC numérique dans un environnement universitaire avec une technologie fortement intégrée. Ce circuit va aussi élargir le champ d'action du laboratoire, fabrication de circuits intégrés, et lui permettre de s'ouvrir à une réflexion plus large et plus profonde dans la réalisation des systèmes électroniques que le laboratoire a acquis depuis sa création en accordant beaucoup d'importance à la mise en application des projets.

Chapitre 5

HNCP-II : une architecture multiprocesseurs homogènes communicants à 16 cœurs en technologie ST 65nm CMOS

5.1 Introduction

Dans le chapitre précédant, nous avons présenté la validation de la brique de base qui consiste dans la fabrication d'un cœur de processeur (HNCP-I) en technologie ST 65nm CMOS. Cette validation nous conduit à l'étape suivante qui est l'HNCP-II.

Ce chapitre présente l'HNCP-II qui possède une architecture multiprocesseurs à 16 cœurs, basée sur le concept des squelettes de parallélisation utilisés par la méthodologie HNCP (SCM, FARM et PIPE). L'intérêt d'une telle architecture est de bénéficier à la fois des avantages de la cible ASIC (performances élevées et faible consommation notamment) ainsi que des squelettes de parallélisation validés sur cible FPGA [1] [27]. Cette architecture constitue une première étape de conception d'architectures multiprocesseurs pour un futur circuit multiprocesseurs (RUN).
Les caractéristiques de cette approche sont liées aux points suivants :

- L'architecture est basée sur le cœur HNCP-I.
- Tous les cœurs ont un accès direct au flot vidéo.
- Tous les cœurs contiennent un système pour le calcul en virgule flottante.
- Tous les nœuds possèdent leurs propres domaine d'horloge. La communication entre nœuds est réalisée via des FIFOs asynchrones, ce qui permet d'avoir des communications fiables.
- Les 16 cœurs peuvent voir leurs moyens de communication configurés en fonction de l'application. Cette configuration est réalisée intégralement de façon logicielle (instruction FSL d'un cycle), puisque la partie matérielle est statique (par nature, la cible étant un ASIC).

– La possibilité de faire du pipeline de squelette, avec une flexibilité dans le choix du nombre d'étages et du nombre de nœuds par étage. Cette flexibilité permet d'ajuster l'implantation afin d'attendre le maximum de performance pour une application donnée.

Le circuit proposé a d'abord été évalué sur une cible FPGA avant d'être synthétisé sur cible ASIC en technologie STMicroelectronics CMOS 65 nm. Le chapitre est organisé comme suit : la section 2 présent l'architecture proposée. L'implantation matérielle est décrite en section 3 et 4. Une validation algorithmique de l'architecture avec quatre algorithmes de base de traitement d'image est abordée en section 5. La section 6 conclut ce chapitre.

5.2 Architecture proposée du HNCP-II

La figure 5.1 donne une vue globale de l'architecture 16 cœurs proposée. Afin d'expliquer nos choix architecturaux, nous allons tout d'abord donner une vue d'ensemble de l'architecture en justifiant le choix de la topologie. Ensuite, nous allons donner des détails sur l'architecture du nœud et le rôle de chaque module. Enfin, nous allons discuter deux aspects importants de cette architecture qui sont le domaine d'horloge et la solution proposée pour résoudre la problématique de la taille de l'image à traiter.

5.2.1 Architecture du circuit

Dans le chapitre 2, nous avons présenté six topologies (anneau, étoile, grille, tore, hypercube et complètement connecté) qui peuvent être considérées dans un contexte d'architectures multiprocesseurs. Toutefois, certaines de ces topologies ne sont pas souhaitables pour plusieurs raisons telle que par exemple la complexité des liens comme c'est le cas des topologies hypercube et complètement connectée ($(n.\log_2(n))/2$ pour la topologie hypercube et $n(n-1)/2$ pour la topologie complètement connecté). Les topologies anneau et étoile ne sont également pas envisageables en raison de leurs faible degré (degré égal à 2 pour la topologie anneau et degré égal à 1 pour tous les nœuds de la topologie étoile à l'exception du nœud central qui a un degré égal à (n-1)). Pour ces raisons de complexité d'implantation (topologies hypercube et complètement connecté) et de faible degré créant un goulot d'étranglement dans la communication (topologies anneau et étoile), ces topologies ont été abandonnées.

Les topologies pouvant être considérées dans un contexte d'architectures avec un grand nombre de nœuds sont la grille et le tore ; en particulier parce qu'elles sont mieux adaptées lors d'une implantation physique que les autres type d'interconnexions (notamment pour les étapes de placement-routage). La topologie choisie dans l'architecture proposée est le tore. Cette topologie permet de réduire les latences de la topologie grille, tout en conservant sa simplicité [77].

Dans l'architecture proposée, tous les nœuds sont identiques, excepté le nœud numéroté **N00** qui communique vers l'extérieur via un bus Wishbone [55]. Quatre périphériques sont connectés au bus Wishbone : une liaison série (UART), un Timer, un périphérique d'entrées/sorties (GPIO) et un contrôleur d'interruption (INTC). Le nœud **N00** est connecté au module "Output Frame Generator" (OFG) qui permet de gérer le flux vidéo sortant. Tous les nœuds du réseau ont accès au même moment au flux vidéo entrant via le module "Input Frame Generator" (IFG). Ce module est configurable à partir de l'extérieur, il peut prendre en charge différentes tailles de l'image. Les modules "Input/Output Frame Generators" génèrent un bus de même format composé de quatre signaux : v_sync, h_sync, v_swap, v_data, comme expliqué dans le chapitre 3. Le module "Clock" permet de générer quatre horloges pour le nœud (le processeur, la FPU, le DMA-Routeur et le module "Frame Grabber" [1]) et une cinquième pour le bus Wishbone et ces périphériques. Le module "Reset" permet de configurer le circuit (mode fonctionnel, mode programmation, mode debug).

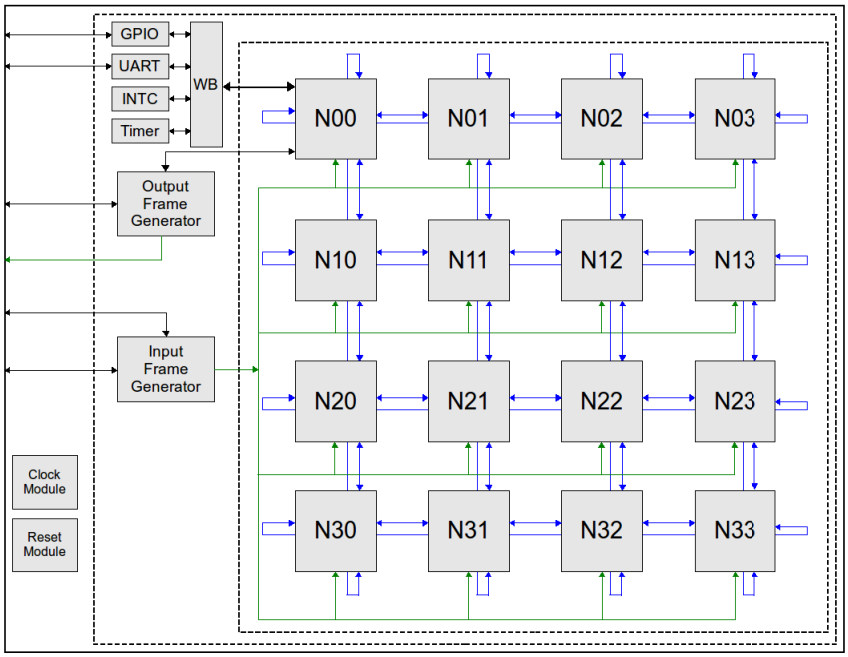


FIGURE 5.1 – Architecture du circuit

5.2.2 Architecture du nœud

L'architecture du nœud est un assemblage de modules développés et précédemment présentés dans le chapitre 3. L'architecture d'un nœud est proposée en figure 5.2 : elle consiste en un cœur de processeur SecretBlaze (modifié) supportant les instructions

1. Les modules "Frame Grabber", "Input Frame Generator" et "Output Frame Generator" ont la même horloge

FSL et un module JTAG IEEE 1149.1. Pour les calculs en virgule flottante, une FPU a été ajoutée au nœud afin d'améliorer les performances. Une unité de contrôle de FPU a également été ajoutée. Elle permet d'implanter diverses fonctions spécifiques de calcul (voir le chapitre 3). La communication avec l'extérieur (entre nœuds) est assurée soit à travers une FIFO asynchrone soit par l'intermédiaire d'un routeur (routage en topologie tore) associé à un contrôleur DMA (Direct Memory Access). Ce dernier permet de recevoir des paquets dans la mémoire interne au nœud en même temps que d'en émettre vers le réseau. Deux multiplexeurs permettent de passer du mode SCM (avec connexion point à point à base de FIFO asynchrone) au mode FARM (utilisant le DMA-Routeur). Pour le flux vidéo, le module "Frame Grabber" permet de choisir la sous-image sélectionnée et de l'écrire dans la mémoire associée "Frame Memory"(FM). L'architecture contient une mémoire locale (LM) (programme) d'une capacité de 16Ko, deux blocs mémoires associés au DMA-Routeur (4Ko sont alloués à la mémoire d'émission (SM) et 4Ko à la mémoire de réception (RM)) et 8Ko pour stocker la sous-image sélectionnée (utilisée par le "Frame Grabber"). Toutes les mémoires sont en accès lecture-écriture par le processeur.

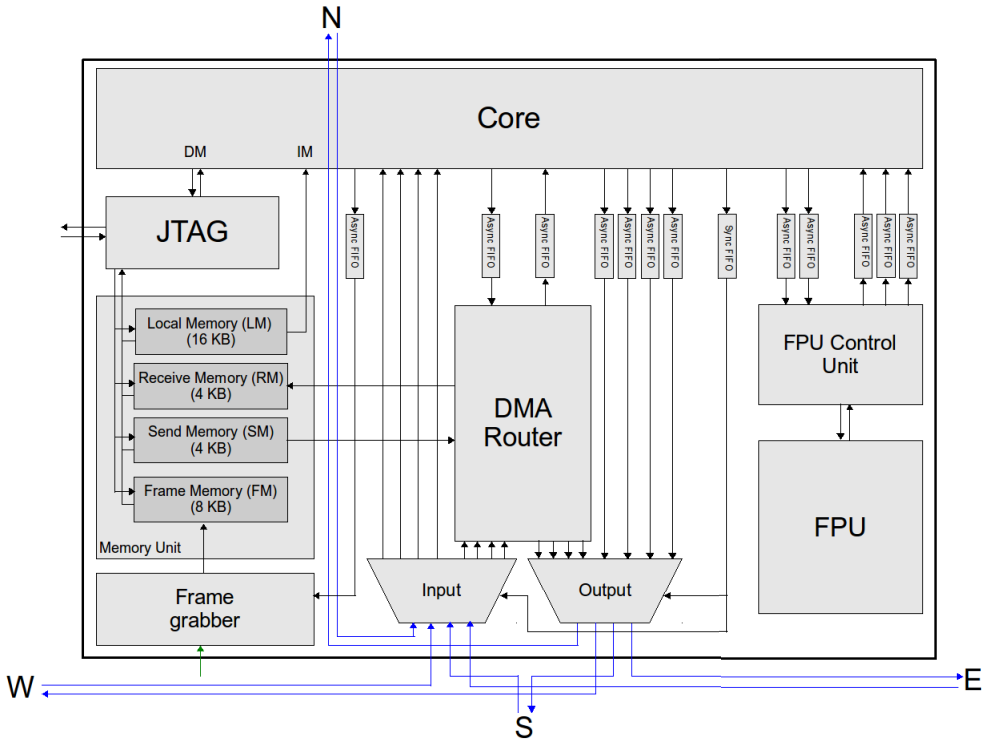


FIGURE 5.2 – Architecture du nœud

5.2.3 Domaine d'horloge

Le domaine d'horloge a été mis en place à cause de la diversité des fréquences de fonctionnement des modules constituant le nœud. Comme présenté dans la sous-section précédemment, les modules nécessitent des échanges constants, ce qui implique que toute donnée transitant d'un module à un autre doit être traitée par un mécanisme de synchronisation. La figure 5.3 montre les domaines d'horloge à l'intérieur du nœud. Les modules de la même couleur font partie du même domaine d'horloge. Le domaine d'horloge est géré grâce aux :

- FIFOs asynchrones dans le cas des communications entre le cœur du processeur et l'unité de contrôle FPU, le DMA-Routeur et le module "Frame Grabber". Les FIFOs asynchrones gèrent aussi l'asynchronisme [2] entre nœuds dans le cas des communications point-à-point.
- Mémoires doubles ports (LM, RM, SM et FM) gèrent l'asynchronisme entre les fréquences du cœur du processeur, du DMA-Routeur et du module "Frame Grabber".

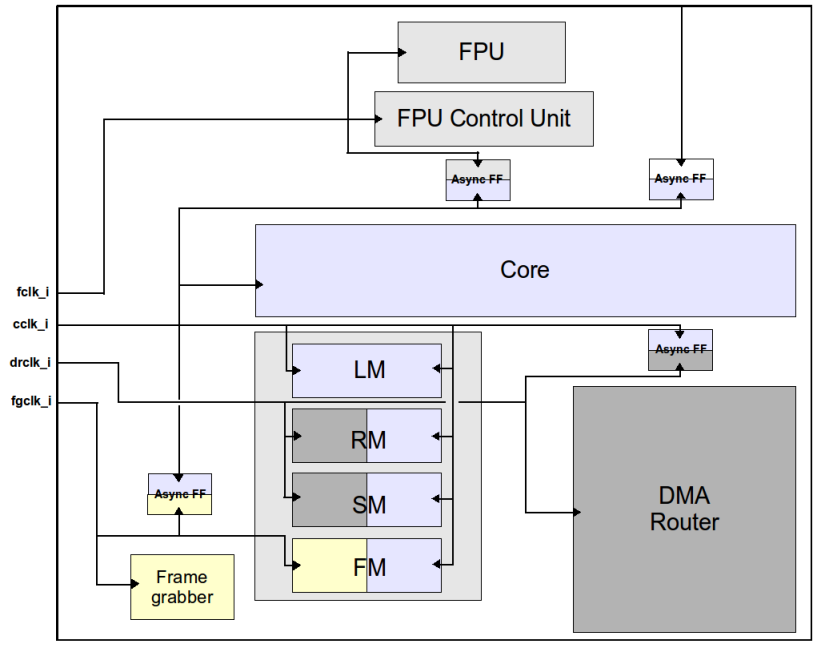


FIGURE 5.3 – Domaine d'horloge

2. Lorsque deux nœuds, en mode SCM, communiquent via des FIFOs asynchrones. L'utilisation des FIFOs asynchrones n'est pas due à un asynchronisme entre les horloges des deux cœurs, mais un moyen d'assurance que la communication est effectuée correctement.

5.2.4 La problématique de la taille de l'image à traiter

La mémoire [3] réservée au stockage d'image (FM) est partagée en deux (SWAP) permettant ainsi de stocker une image de taille 4Ko ce qui fait pour une image codée en niveau de gris 4096 pixels (64x64 pixels dans le cas d'une image carrée). En une seule fois, l'architecture peut traiter une image de 16 x 4096 pixels (256x256 pixels dans le cas d'une image carrée).
Pour traiter une image plus grande, cette contrainte doit être prise en compte. Par exemple, pour traiter une image de 1Mo (1024x1024 pixels en niveau de gris) en utilisant tous les nœuds de l'architecture. Celle-ci est divisée en 4 sous-images de taille 256x256 et envoyée en quatre fois via le module OFG. Une fois la première sous-image traitée, la deuxième est envoyée et ainsi de suite. Le nombre d'envoi, qui est égal au nombre de sous-images de l'image globale, peut être exprimé par l'expression suivante :

$$N_{envoi} = \frac{S_{globale}}{N_{noeuds} \times S_{noeud}} \tag{5.1}$$

Avec N_{envoi} le nombre d'envoi, $S_{globale}$ la taille de l'image globale en octets (nombre de pixels multiplié par 3 si c'est une image couleur ou multiplier par 1 octet si c'est une image en niveau de gris), N_{noeuds} le nombre de nœud utilisé (maximum : 16 nœuds) et S_{noeud} la taille d'image à traiter par un seul nœud (maximum : 4096 octets).

5.3 Implantation sur FPGA

Dans cette section, nous allons donner les résultats d'implantation matérielle (fréquence, consommation et surface) sur cible FPGA. L'architecture a été synthétisé et placé routé utilisant le Virtex-6 XC6VLX760 (speedgrade -1) [4].
Les fréquences d'entrées de l'architecture sur la cible FPGA sont de 200MHz pour le core, le FPU, le "FPU Control Unit" et le DMA-Router, de 100MHz pour le module "Frame Grabber", le bus Wishbone et les périphériques (GPIO, UART, INTC and Timer).
La consommation statique est de $4,73W$ et la consommation total (statique + dynamique) est de $9,76W$.
L'utilisation des ressources matérielles des composants constituant le nœud (présenté sur la figure 5.2) sont 7193 Slice Reg, 9901 LUT, 347 LUTRAM et 16 BRAM. Les ressources matérielles occupées par un nœud constituent 4,96% des Slices (5887 Slices) sur la totalité des ressources disponibles sur ce FPGA. Pour ce qui est de la totalité de l'architecture (présentée sur la figure 5.1), l'utilisation des ressources matérielles sont

3. La restriction de la taille d'image à 4 Ko est basé sur notre expérience d'implantation effectuée dans l'HNCP-I. Les résultats d'implantation du premier RUN (avec une taille mémoire de 4 Ko) ont montré que celle-ci couvre la majeure partie de la zone active du circuit. Dans l'optique de réalisation du HNCP-II et selon nos contraints budgétaires, la taille mémoire choisie nous paraît réaliste.

4. Le plus grand FPGA disponible dans notre laboratoire est le XC6VLX240T, ce dernier ne peut contenir l'architecture. Nous rappelons que la technologie de fabrication de cet FPGA est la 40 nm tandis que celle utilisée sur cible ASIC est la 65 nm.

115857 Slice Reg, 158835 LUT, 5580 LUTRAM et 256 BRAM. Les ressources matérielles occupées par cette architecture constituent 78,35% en Slices (92899 Slices) sur la totalité des ressources disponibles sur cet FPGA.

5.4 Implantation sur ASIC

Dans cette section, nous allons donner les résultats d'implantation matérielle (fréquence, surface et consommation) sur cible ASIC. Le flot et les outils de conception utilisés sont les mêmes que ceux du HNCP-I (voir la sous-section 4.6.1 du chapitre 4).

5.4.1 Résultats après l'étape de synthèse logique

Les fréquences d'entrées de l'architecture sur la cible ASIC sont de 400MHz pour le core, le FPU, le "FPU Control Unit" et le DMA-Router, de 200MHz pour le module "Frame Grabber", le bus Wishbone et les périphériques (GPIO, UART, INTC et Timer). La limite en fréquence de 400MHz est due à la conception du module FPU dont nous n'étions pas maître.
Pour ce qui est de la surface et de la consommation statique au **niveau nœud**, le tableau 6.1 présente les résultats de synthèse logique en terme de nombre de cellules, de surface et de consommation statique pour les composantes constituant le nœud présenté sur la figure 5.2. La surface et la consommation statique totale d'un nœud avant placement routage sont respectivement de $1,54mm^2$ et $1,90mW$.

Modules	**Nombre de cellules**	**Surface (um^2)**	**Consommation statique (μW)**
Memory Unit	1893 (4,41 %)	1300975 (84,45%)	1814,33 (95,38%)
Core	11307 (26,33%)	54188 (3,52%)	17,95 (0,94%)
DMA-Router	6219 (14,48%)	43259 (2,81%)	15,11 (0,79%)
FPU	8014 (18,66%)	38256 (2,48%)	13,44 (0,71%)
FPU Control Unit	498 (1,16%)	2408 (0,16%)	0,85 (0,04%)
Frame Grabber	1222 (2,85%)	6173 (0,40%)	2,20 (0,12%)
JTAG	1691 (3,94%)	10880 (0,71%)	4,60 (0,24%)
Muxs, Buses,...	12097 (28,17%)	84280 (5,47%)	33,79 (1,78%)
Total	**42941 (100%)**	**1540419 (100%)**	**1902,27 (100%)**

TABLE 5.1 – Résultats d'implantation du nœud sur ASIC en technologie CMOS 65 nm

Pour ce qui est de la surface et de la consommation statique au **niveau circuit**, le tableau 6.2 présente les résultats de synthèse logique en terme de nombre de cellules, de surface et de consommation statique pour l'architecture présentée sur la figure 5.1. La surface et la consommation statique totale de cette architecture sont respectivement de $24,66mm^2$ et $30,44mW$, avant ajout des cellules d'entrées/sorties et avant placement-routage.

Modules	**Nombre de cellules**	**Surface (μm^2)**	**Consommation statique (μW)**
16 Nodes	688627 (99,70%)	24658103 (99,96%)	30443,59 (99,99%)
Wishbone bus	112 (0,02%)	360 (0,001%)	0,09 (0,0003%)
TIMER	571 (0,08%)	2968 (0,01%)	1,04 (0,003%)
UART	280 (0,04%)	1303 (0,005%)	0,47 (0,001%)
INTC	183 (0,03%)	794 (0,003%)	0,28 (0,001%)
GPIO	63 (0,01%)	334 (0,001%)	0,12 (0,0004%)
INPUT Frame Gen.	399 (0,06%)	2060 (0,01%)	0,78 (0,002%)
OUTPUT Frame Gen.	397 (0,06%)	2045 (0,01%)	0,76 (0,002%)
Clock,Reset,...	37 (0,005%)	220 (0,0009%)	0,15 (0,0005%)
Total	**690669 (100%)**	**24668187 (100%)**	**30447,28 (100%)**

TABLE 5.2 – Résultats d'implantation du circuit sur ASIC en technologie 65 nm

Pour ce qui est de la consommation dynamique, le synthétiseur (Design Compiler) donne un résultat de consommation interne des cellules (Cell Internal Power) de $2,94W$ à une fréquence de 200MHz. A 400MHz, le synthétiseur donne une consommation interne des cellules de $5,89W$.

5.4.2 Résultats après l'étape de placement-routage

La mémoire prend la plus grande partie de la zone active du circuit, comme on peut voir sur la figure 6.2. La vue à droite appelée vue Amoeba affiche le contour des modules et sous-modules après placement montrant ainsi la localisation physique des modules (la mémoire de chaque nœud est encadrée et numérotée). La taille de la mémoire choisie est équivalente à 2048 blocs de (256x8bits). La surface totale du circuit est de $45,88mm^2$ (deltaX=$5,526mm$ et deltaY=$8,304mm$). Le nombre d'entrées/sorties est de 326 (incluant 48 plots d'alimentation). La fréquence estimée lors de la synthèse logique est maintenue (400MHz).
L'analyse en puissance donne une consommation statique totale de $31,75mW$ (les mémoires consomment $28.98mW$ ce qui représente 91.28% de la consommation statique du circuit) et un maximum de consommation totale de $9,53W$. La puissance dynamique a été calculée à l'aide de l'expression CV^2F en considérant le cas critique d'un taux d'activité égal à 1 (C : capacité extraite ($19,63nF$), V : tension d'alimentation ($1.1V$) et F : fréquence d'entrée (400MHz)) [5].

Le tableau 5.3 montre la consommation statique du circuit en fonction des librairies utilisées. Comme mentionné précédemment, les mémoires (DPHS) consomment 91,28%, même si elles ne représentent que 0,23% du nombre total de cellule. Les cellules

5. Cette consommation élevée peut être expliquée par l'utilisation dans la formule de calcul de la puissance dynamique :
- d'un taux d'activité égale à 1 or celui-ci est généralement inférieur à 0.5,
- de la capacité totale extraite au lieu de la capacité commutée extraite.

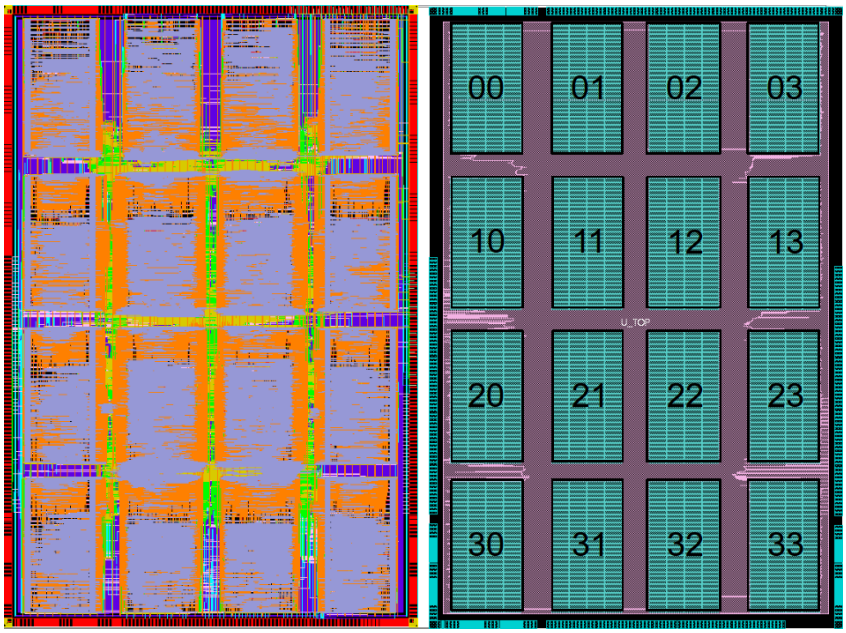


FIGURE 5.4 – Layout du circuit vue physique (à gauche) vue Amoeba (à droite)

standards (CORE65LPSVT et CLOCK65LPSVT) consomment 7,28% en représentant 98,04% du nombre total de cellules.

Librairies	**Nombre de cellule**	**Consommation statique** (mW)
DPHS	2048 (0,23%)	28,98 (91,28%)
CORE65LPSVT	860195 (98,04%)	2,31 (7,28%)
CLOCK65LPSVT	13669 (1,56%)	0,05 (0,15%)
IO65LP_SF_BASIC_50A	1248 (0,14%)	0,32 (1,01%)
IO65LPHVT_SF_1V8_50A	278 (0,03%)	0,09 (0,28%)

TABLE 5.3 – Résultats de nombre de cellules et de consommation statique après placement-routage par librairies

5.5 Validations algorithmiques

Dans cette section, nous présentons la validation de l'architecture proposée sur les trois squelettes de parallélisation supportés par celle-ci en utilisant quatre exemples simples d'algorithmes de traitement d'images.
Les deux premiers algorithmes présentés dans le tableau 5.4, histogramme et seuillage adaptatif, permettent de valider le squelette SCM. La différence entre les deux algo-

rithmes consiste en leurs sorties et leurs complexités (tab. 5.4). L'objectif est de montrer l'impact de la communication et de la complexité de l'algorithme sur la performance de l'architecture.
Les deux derniers algorithmes (tab. 5.4), mise en correspondance de primitives et Harris et Stephen [78], ont été implantés précédemment sur FPGA en utilisant l'approche HNCP dans les thèses [1] et [27]. L'algorithme de mise en correspondance de primitives permet de valider le squelette FARM. Cet algorithme est aussi utilisé avec l'algorithme de Harris et Stephen afin de valider le squelette PIPE.

#	Algorithmes	Entrées	Sorties	Étape de calcul	Type de parallélisme	Squelette adéquat
1	**Calcul d'Histogramme**	Image	Vecteur de 256 données	1 étape simple	Parallélisme de données statique	SCM
2	**Seuillage adaptatif**	Image	Image	3 étapes simples	Parallélisme de données statique	SCM
3	**Mise en correspondance de primitives**	Image et points d'intérêts	Points d'intérêts et scores	3 étapes complexes	Parallélisme de données dynamique	FARM
4	**Harris et Stephen**	Image	Points d'intérêts	2 étapes complexes statique	Parallélisme de données	SCM
5	**L'algorithme #4 suivi de l'algorithme #3**	Image	Points d'intérêts et scores	2 étapes suivis de 3 étapes	2 types de parallélisme pipelinés	PIPE

TABLE 5.4 – Algorithmes utilisés pour la validation algorithmiques de l'architecture

5.5.1 Configurations architecturales d'évaluations

Dans l'architecture proposée, nous avons un grand nombre de combinaisons d'implantations possibles sachant que le nœud N00 doit être intégré dans toutes les implantations pour la raison évoquée précédemment (voir la section 5.2.1). La figure 6.8 présente cinq configurations d'implantations algorithmiques dans l'architecture proposée (dans (a) l'algorithme est implanté dans un seul nœud ; dans (e) l'algorithme est implanté dans tous les nœuds). Ces implantations montrent le degré de flexibilité de l'architecture proposée, elle sont utilisées pour illustrer les performances de l'architecture.
Certaines de ces implantations ne sont pas intéressantes du fait de la faible communication (degré) entre nœuds (voir chapitre 2 : 2.3 Les topologies). Par exemple pour une architecture à 4 nœuds, prendre N00, N01, N02 et N03 ou N00, N10, N20 et N30 ou encore N00, N01, N10 et N11 donne le même résultat, car tous les nœuds ont le même

degré (qui est 2). Par contre, si on prend par exemple les 4 nœuds N00, N01, N11 et N12, le nœud N00 et N12 sont de degré 1 et l'étape de fusion a un seul chemin qui est du N12 vers N11 puis vers N01 puis vers N00.

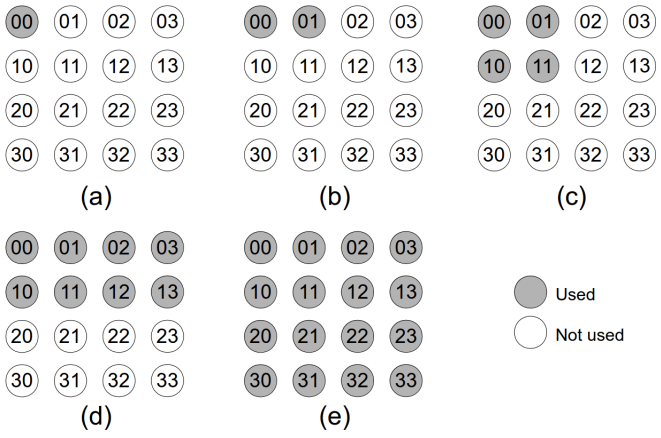


FIGURE 5.5 – Configurations architecturales d'évaluations

5.5.2 Validation du squelette SCM sur l'architecture proposée

La figure 5.6 montre le squelette SCM tel qu'il est implanté dans l'architecture proposée. Grâce au module "Frame Grabber", chaque processeur possède un accès direct aux données d'entrées. Le rôle de la fonction "Split" est la récupération des données d'entrées en configurant le module "Frame Grabber". Ce qui permet à tous les processeurs, après une étape de synchronisation, de commencer l'étape de "Compute" en même temps.

5.5.2.1 Principe de fonctionnement en mode SCM

Chaque nœud sélectionné est mis en mode SCM, c'est-à-dire que la communication est réalisée avec des liens point-à-point via des FIFOs asynchrones comme le montre la figure 5.2. La fonction de division "split" implantée dans chaque nœud sélectionné permet de contrôler le module "Frame Grabber" et ainsi de récupérer la sous-image en temps réel. Il faut noté que lorsque l'image i est écrite dans la mémoire, c'est l'image i-1 qui est en cours de traitement, ce qui rend la phase de "split" transparente. Chaque nœud prend la sous-image sélectionnée afin d'exécuter l'algorithme. Une fois l'étape de traitement finie, le résultat est envoyé au nœud N00, grâce à la fonction de fusion "Merge" [6].

6. Le nœud N00 est par la suite transférer ($T_{transfer}$) le résultat de traitement soit au module OFG (via une liaison point-à-point FIFO asynchrone pour affichage) soit au nœud d'un deuxième étage de pipeline afin d'effectuer la chaîne complète de traitement.

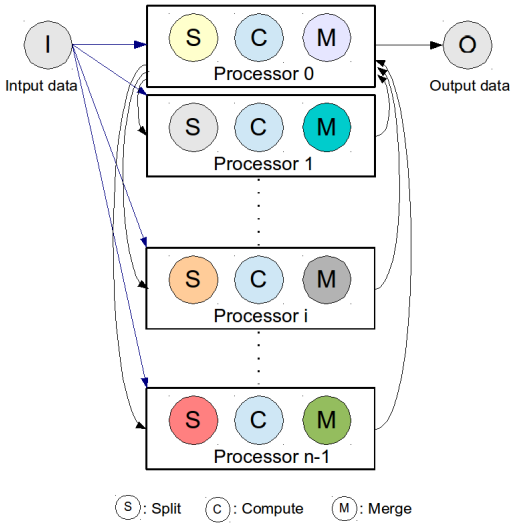


FIGURE 5.6 – Représentation graphique de la version implantée du squelette SCM

5.5.2.2 Algorithme de calcul d'Histogramme

Pour une image codée sur L niveau de gris (une image ayant 8 bits/pixel, L = 255), l'histogramme en niveau de gris de l'image est définie par une fonction $h(g)$ qui a comme valeur, pour chaque niveau d'intensité g ϵ [0 ... L], le nombre de pixels de l'image ayant une intensité égale à g.

$$h(g) = N_g \tag{5.2}$$

Avec N_g est le nombre de pixels de l'image qui ont la même intensité égale à g.

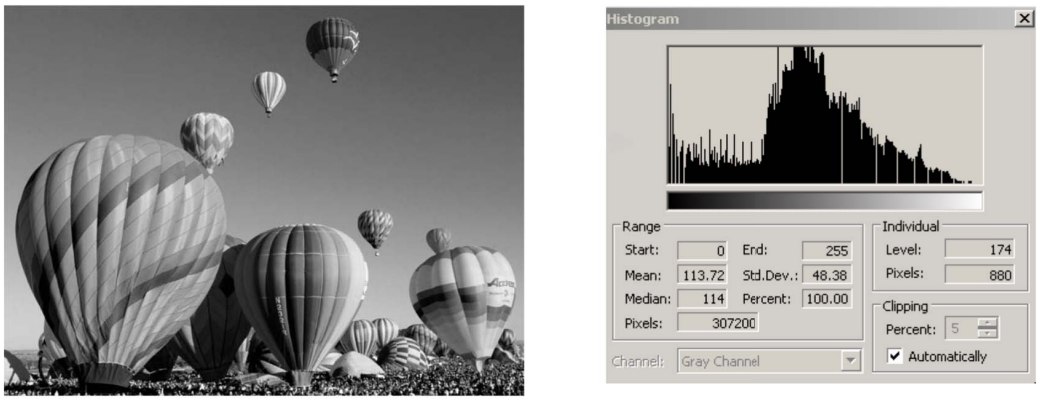


FIGURE 5.7 – Exemple d'un calcul d'histogramme pour une image en niveaux de gris [79]

5.5.2.3 Résultats d'exécution

Le tableau 6.9 présente le résultat d'exécution [7] de l'algorithme du calcul d'histogramme sur l'architecture proposée. Les résultats sont présentés en temps d'exécution et accélération en fonction de la taille de l'image d'entrée et le nombre de nœuds utilisés dans l'implantation.

	Taille de l'image	**Implantations**				
		a (1)	**b (2)**	**c (4)**	**d (8)**	**e (16)**
Temps d'exécution (ms)	64x64	0,13	0,08	0,07	0,08	0,10
	256x256	2,13	1,12	0,62	0,37	0,22
	512x512	8,52	4,51	2,49	1,48	0,88
	1024x1024	34,10	18,03	9,96	5,92	3,54
	2048x2048	136,41	72,15	39,84	23,69	14,18
Accélération	64x64	1,00	1,53	1,88	1,57	1,25
	256x256	1,00	1,89	3,42	5,74	9,59
	512x512	1,00	1,89	3,42	5,75	9,61
	1024x1024	1,00	1,89	3,42	5,75	9,61
	2048x2048	1,00	1,89	3,42	5,75	9,61

TABLE 5.5 – Temps d'exécution et accélération de l'algorithme du calcul d'histogramme sur l'architecture proposée

• **Temps d'exécution**

Le temps total d'exécution peut être exprimé comme suit :

$$T_{total} = T_{split} + T_{compute} + T_{merge} + T_{transfer} \tag{5.3}$$

Comme mentionné précédemment, le temps (T_{split}) est transparent car lorsque le module "Frame Grabber" est en cours d'écriture de l'image courante, le processeur est en cours de traitement de l'image précédente. Le ($T_{transfer}$) est le temps nécessaire pour transférer les résultats d'un étage de pipeline à l'autre ou vers le module OFG. Ce temps $T_{transfer}$ est le même pour les cinq implantations quelque soit la taille de l'image, il est égal à $7,04\mu s$ (vecteur de 256 données de 32 bits), car le résultat d'un calcul d'histogramme est toujours fixe.

Le temps ($T_{compute}$ + T_{merge}) est diffèrent selon la taille de l'image traitée par nœud et le nombre de nœuds utilisés. Le temps total d'exécution peut être exprimé comme suit :

$$T_{total} = T_{compute} + T_{merge} + T_{transfer} \tag{5.4}$$

7. Les résultats des temps d'exécutions pour les tailles d'images 1024x1024 et 2048x2048 ont été estimés à partir des résultats des temps d'exécutions des images inférieurs (pour les quatre algorithmes présentés dans ce chapitre).

Dans le premier cas où la taille d'image est de 64x64, le calcul se fait en une seule fois (c'est-à-dire que le N_{envoi}=1) [8] quelque soit le nombre de nœuds (même dans le cas de l'implantation (a)). Dans le deuxième cas où la taille d'image est de 256x256, le calcul se fait en une seule fois pour l'implantation (e), en deux fois pour l'implantation (d), en quatre fois pour l'implantation (c), en huit fois pour l'implantation (b) et en seize fois pour l'implantation (a). Le temps total d'exécution peut être exprimé comme suit :

$$T_{total} = (N_{envoi} \times (T_{compute} + T_{merge})) + ((N_{envoi} - 1) \times T_{add}) + T_{transfer} \quad (5.5)$$

Par exemple, pour une image 512x512 sur l'architecture (e), l'image globale est divisée en 4 sous-images ($N_{envoi} = 4$). Lorsque la première sous-image (256x256) est traitée, le vecteur résultat est sauvegardé dans le nœud N00. La deuxième sous-image est alors envoyée puis traitée et enfin les deux vecteurs de résultats sont additionnés (temps d'addition (T_{add}) de deux vecteurs est égal à $6,4\mu s$) afin d'obtenir le résultat du calcul d'histogramme pour les deux sous-images. Ce processus se répète jusqu'à l'addition du quatrième résultat de la quatrième sous-image.

• **Accélération**

L'accélération est définie comme le rapport entre le temps d'exécution de l'algorithme de façon séquentiel (c'est-à-dire avec un seul nœud, ce qui correspondant à l'implantation (a)) et le temps d'exécution de l'algorithme de façon parallèle (c'est-à-dire sur des architectures à deux nœuds ou plus), ce qui correspond dans le tableau aux implantations (b), (c), (d) et (e). Le calcul d'accélération peut être exprimé comme suit :

$$S_n = \frac{T_1}{T_n} \quad (5.6)$$

L'accélération idéale théorique est obtenue lorsque S_n = n. Pour le cas d'une image 64x64, le tableau 6.9 montre une faible accélération avec un meilleur résultat obtenu pour l'implantation (c). Nous pouvons dire que c'est l'implantation idéale pour cet algorithme avec cette taille d'image. Ce résultat est dû au fait que le temps de traitement est très faible dans le cas de cet algorithme. Il s'explique aussi par le fait que le temps de l'étape de fusion augmente avec le nombre de nœuds (tous les résultats doivent être envoyés au nœud 00).

A partir de la taille d'image 256x256, l'accélération s'améliore pour atteindre 9,59 dans la configuration (e). Cette accélération est constante quelque soit la taille de l'image. Ce résultat s'explique à travers les formules 5.5 et 5.6.

Par exemple pour la configuration (c) avec la taille d'image 256x256, le nombre d'envoi est égal à 16 fois pour le temps d'exécution séquentiel (T_1) et à 4 fois pour le temps d'exécution parallèle (T_4).

8. Pour rappel, la taille de la mémoire image (FM) par nœud est de 4Ko.

$$S_4 = \frac{T_1}{T_4} = \frac{(16 \times (T_{compute} + T_{merge})) + ((16-1) \times T_{add}) + T_{transfer}}{(4 \times (T_{compute} + T_{merge})) + ((4-1) \times T_{add}) + T_{transfer}} \tag{5.7}$$

$$S_4 = \frac{16 \times (T_{compute} + T_{merge} + T_{add} - \frac{T_{add}}{16} + \frac{T_{transfer}}{16})}{4 \times (T_{compute} + T_{merge} + T_{add} - \frac{T_{add}}{4} + \frac{T_{transfer}}{4})} \tag{5.8}$$

Si on prend, maintenant une image de 512x512, le nombre d'envois est égal à 64 dans le cas de l'exécution séquentielle et de 16 dans le cas d'exécution parallèle ce qui fait 64/16 (de même pour 1204x1024 (256/64) et 2048x2048 (1024/256)).

$$S_4 = \frac{T_1}{T_4} = \frac{64 \times (T_{compute} + T_{merge} + T_{add} - \frac{T_{add}}{64} + \frac{T_{transfer}}{64})}{16 \times (T_{compute} + T_{merge} + T_{add} - \frac{T_{add}}{16} + \frac{T_{transfer}}{16})} \tag{5.9}$$

Chaque fois que l'image est plus grande, les rapports T_{add}/N_{envoi} et $T_{transfer}/N_{envoi}$ tendent vers 0. Nous obtenons ainsi des accélérations constantes jusqu'au troisième chiffre après la virgule.

5.5.2.4 Algorithme du seuillage adaptatif

Le seuillage d'une image consiste à binariser une image en niveau de gris en vue par exemple d'une segmentation. L'algorithme proposé [80] utilise la valeur d'un seuil qui est paramétré dynamiquement afin de rendre compte de l'intensité locale des différentes zones de l'image. Cet algorithme du seuillage adaptatif se décompose en trois étapes (fig. 5.8) :

- Calcul de l'image intégrale.
- Calcul des cordonnées du coin supérieur gauche (x1, y1), et du coin inférieur droit (x2, y2) afin de calculer la somme pour le rectangle.
- Effectuer la comparaison afin de classer le pixel soit noir ou blanc.

Une image intégrale est utilisé chaque fois que nous souhaitons calculer la somme d'une fonction de pixels (f(x, y) avec x et y nombres réels) sur une zone rectangulaire de l'image. Pour calculer l'image intégrale, nous stockons à chaque endroit, I (x, y), la somme de toutes les f(x, y) des termes à gauche et au-dessus du pixel (x, y) (fig. 5.8). Ceci est accompli en un temps linéaire en utilisant l'équation suivante pour chaque pixel :

$$I(x, y) = f(x, y) + I(x-1, y) + I(x, y-1) - I(x-1, y-1) \tag{5.10}$$

La somme de la fonction pour tout rectangle avec le coin supérieur gauche (x1, y1), et le coin inférieur droit (x2, y2) peut être calculée en utilisant l'équation suivante :

$$f(x, y) = \sum_{x=x1}^{x2} \sum_{y=y1}^{y2} I(x2, y2) - I(x2, y1-1) - I(x1-1, y2) + I(x1-1, y1-1) \tag{5.11}$$

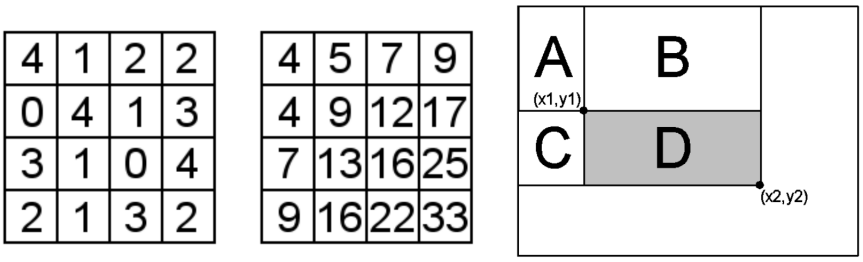


FIGURE 5.8 – Calcul de l'image intégrale [80]

L'image de gauche de la figure 5.8 présente l'image d'entrée, celle du centre présente l'image intégrale calculée et celle de droite présente l'utilisation de l'image intégrale pour calculer la somme sur un rectangle D. Le calcul de la somme de f(x, y) sur le rectangle D en utilisant l'équation 2 est équivalent au calcul de la somme sur les rectangles (A + B + C + D) - (A + B) - (A + C) + A. L'image à droite de la figure 5.9 présente le résultat d'un seuillage adaptatif.

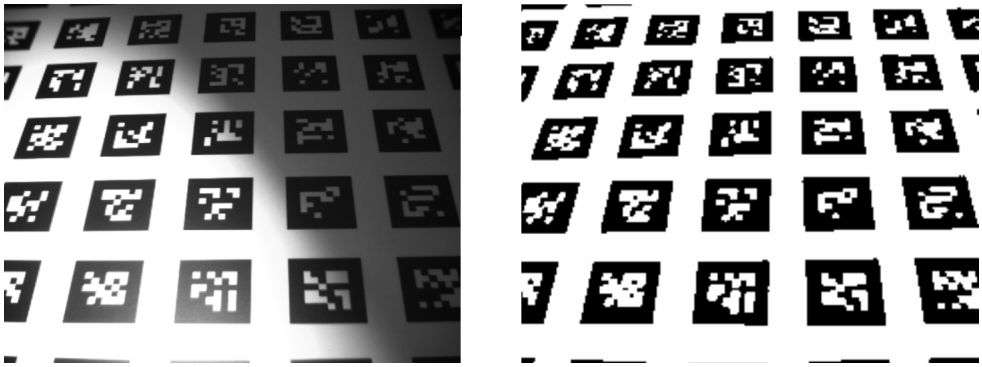


FIGURE 5.9 – Représentation graphique d'un résultat de seuillage adaptatif

5.5.2.5 Résultats d'exécution

Le tableau 6.10 présente le résultat d'implantation de l'algorithme du seuillage adaptatif sur l'architecture proposée. Les résultats sont présentés en temps d'exécution et accélération en fonction de la taille de l'image d'entrée et le nombre de nœuds utilisés dans l'implantation.

- **Temps d'exécution**

Dans le cas de l'algorithme du seuillage adaptatif, le résultat final (contrairement à l'algorithme du calcul d'histogramme) n'est pas la simple addition des sous-résultats pour chaque sous-image. Une fois le résultat prêt pour une sous-image, celui-ci est

	Taille de l'image	**Implantations**				
		a (1)	**b (2)**	**c (4)**	**d (8)**	**e (16)**
Temps d'exécution (ms)	64x64	0,79	0,48	0,33	0,26	0,22
	256x256	12,67	7,94	5,52	4,32	3,71
	512x512	50,69	31,77	22,11	17,28	14,86
	1024x1024	202,78	127,09	88,45	69,13	59,46
	2048x2048	811,15	508,39	353,81	276,52	237,87
Accélération	64x64	1,00	1,62	2,34	3,04	3,56
	256x256	1,00	1,59	2,29	2,93	3,41
	512x512	1,00	1,59	2,29	2,93	3,41
	1024x1024	1,00	1,59	2,29	2,93	3,41
	2048x2048	1,00	1,59	2,29	2,93	3,41

TABLE 5.6 – Temps d'exécution et accélération de l'algorithme du seuillage adaptatif sur l'architecture proposée

transféré. Le temps total d'exécution peut être exprimée comme suit :

$$T_{total} = N_{envoi} \times (T_{compute} + T_{merge} + T_{transfer}) \tag{5.12}$$

Le temps de fusion ((T_{merge}) est très pénalisant dans le cas de cet algorithme et ceci pour deux raisons : la quantité de données à transmettre (une image de taille 64x64 pour chaque nœud) et le temps d'attente lorsque le nœud N00 transfert le résultat final comme expliqué précédemment. L'implantation (e) fournie les meilleurs temps d'exécutions pour toutes les tailles d'image même pour la 64x64 contrairement à l'algorithme du calcul d'histogramme, car le temps de traitement ($T_{compute}$) est plus important dans cet algorithme.

- **Accélération**

L'accélération diminue à chaque fois que le nombre de processeurs augmente. L'accélération du système passe à 3,4 pour l'implantation (e), ce qui est très faible même si cette implantation nous permet d'avoir un temps d'exécution intéressant par rapport à la solution non-parallélisée. Idéalement l'accélération devrait être égale à 16, mais avec l'impact de la communication cette valeur ne peut être atteinte. Toutefois, l'algorithme de seuillage adaptatif n'est pas très gourmand en calcul, nous ajoutons a cela l'aspect communication (pire cas qu'on peut avoir ç-à-d que chaque nœud transmet une image de 64x64 comme résultat de traitement).

5.5.3 Validation du squelette FARM sur l'architecture proposée

La figure 5.10 montre le squelette FARM tel qu'il est implanté dans l'architecture proposée. Par rapport au squelette initial de la méthodologie HNCP, l'amélioration apportée à ce squelette est identique à celle apportée au squelette SCM : tous les processeurs possèdent un accès direct aux données d'entrées.

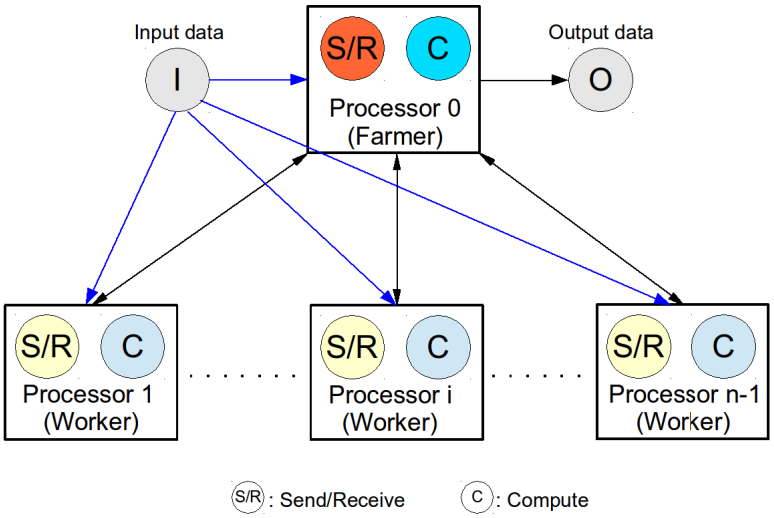


FIGURE 5.10 – Représentation graphique de la version implantée du squelette FARM

5.5.3.1 Principe de fonctionnement en mode FARM

Chaque nœud sélectionné est mis en mode FARM (c'est-à-dire que la communication est réalisée via le DMA-Routeur comme le montre la figure 5.1) au début de chaque programme implanté dans chaque nœud. Dans toutes les implantations effectuées dans cette section, le nœud N00 est considéré comme le maître et les autres sont considérés comme des esclaves (le choix du nœud N00 peut être changé, car n'importe quel nœud peut réaliser la fonction de maître). Le nœud N00 (maître) envoie les points (cordonnées (x, y)) pour la mise en correspondance aux esclaves grâce au DMA-Routeur. Une fois le traitement de chaque nœud fini, le nœud N00 récolte les résultats des traitements des esclaves et transfert le résultat soit au module OFG soit au nœud d'un deuxième étage de pipeline.

5.5.3.2 Algorithme de mise en correspondance de primitives

L'algorithme de mise en correspondance est basé sur un schéma classique qui consiste en l'appariement de primitives après leurs détection (par la méthode de Harris et Stephen, voir plus loin). Pour chacune des primitives, nous calculons un descripteur local qui permet d'obtenir un score de ressemblance entre deux primitives. L'appariement est effectué par un algorithme qui retient les primitives maximisant ce score de ressemblance et exploitant des contraintes de voisinage. En pratique, les images courantes peuvent être mises en correspondance (matched) en prenant soit l'image précédente soit une image mémorisée en tant qu'image de référence (appelée aussi image clé). Le score de ressemblance est calculé en effectuant une corrélation croisée centrée et normalisée (ZNCC) sur un voisinage de 11×11 pixels. Cet algorithme a été utilisé directement

(déjà écrit en langage C) à partir des travaux de thèse effectués dans notre équipe [1] et [27]. La figure 5.11 présente une mise en correspondance des primitives entre deux images.

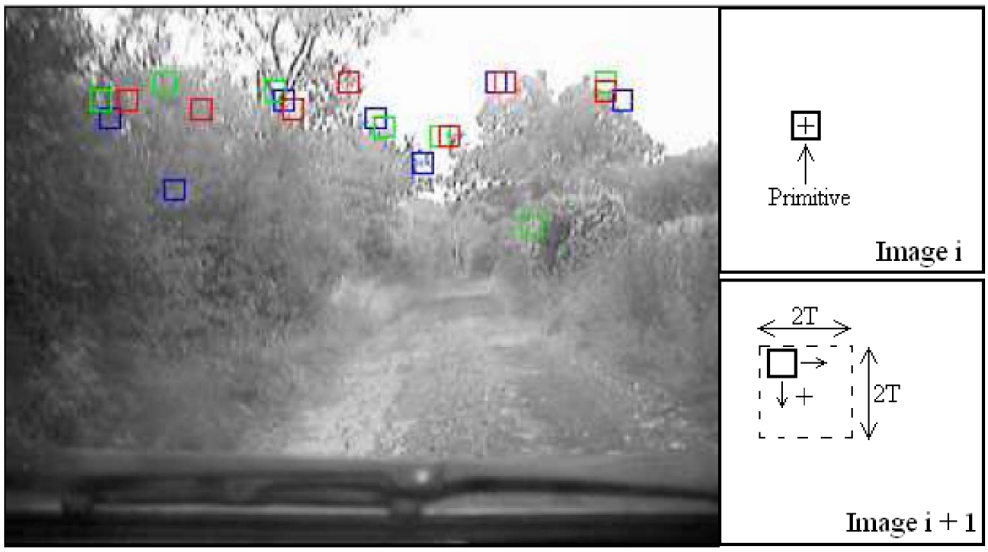


FIGURE 5.11 – Exemple de primitives sélectionnées sur une séquence d'images réelle (à gauche). Schéma de la mise en correspondance des primitives entre deux images (à droite) [25]

5.5.3.3 Résultats d'exécution

Le tableau 6.11 présente le résultat d'implantation de l'algorithme de mise en correspondance sur l'architecture proposée. Les résultats sont présentés en temps d'exécution et accélération en fonction de la taille de l'image d'entrée et le nombre de nœuds utilisés dans l'implantation. Les résultats sont présentées en prenant en compte l'hypothèse suivante : un (et un seul) point d'intérêt est à mettre en correspondance (matching) pour chaque image de taille 64x64 pixels. Cela signifie que pour une image de taille 256x256, le nombre de point à mettre en correspondance est de 16 points.

- **Temps d'exécution**

Les points à mettre en correspondance sont généralement fournis par un autre algorithme en amont de celui-ci. Dans cette application, nous avons fixé les points (cordonnées (x,y)) dans le nœud maître (N00), ce dernier distribue un point à mettre en correspondance pour chaque taille d'image de 64x64. Par exemple pour une taille d'image de 256x256, nous avons 16 points à mette en correspondance. dans le cas d'une implantation à 16 nœuds (e), ce calcul se fait en une seule fois. L'estimation du temps total d'exécution peut être exprimée comme suit :

$$T_{total} = N_{envoi} \times (T_{send\_points} + T_{compute} + T_{receive\_results} + T_{transfer}) \qquad (5.13)$$

	Taille de l'image	Implantations				
		a (1)	b (2)	c (4)	d (8)	e (16)
Temps d'exécution (ms)	64x64	0,58	-	-	-	-
	256x256	9,31	4,66	2,33	1,16	0,58
	512x512	37,25	18,64	9,32	4,67	2,34
	1024x1024	149,02	74,57	37,30	18,68	9,36
	2048x2048	596,10	298,29	149,23	74,73	37,47
Accélération	64x64	1,00	-	-	-	-
	256x256	1,00	1,99	3,99	7,97	15,90
	512x512	1,00	1,99	3,99	7,97	15,90
	1024x1024	1,00	1,99	3,99	7,97	15,90
	2048x2048	1,00	1,99	3,99	7,97	15,90

TABLE 5.7 – Temps d'exécution et accélération de l'algorithme de mise en correspondance sur l'architecture proposée

Nous obtenons de très bons résultats pour les temps d'exécutions de cet algorithme. Celui-ci utilisé dans une application de stabilisation d'image s'effectue pour une image 512x512 en environ 2ms pour l'implantation (e). Ce qui pour un capteur standard (40 ms) laisse suffisamment de temps pour un traitement en aval.

• **Accélération**

Le tableau 6.11 montre une accélération proche de l'accélération théorique pour toutes les implantations. Ce résultat est dû au fait que :

- les temps d'émissions et de réceptions des données entre le maître et les esclaves sont très faibles,
- les faibles quantités de données à émettre (2 données de 32 bits : cordonnées(x, y)) et à recevoir (5 données de 32 bits : cordonnées(x1, y1), nouvelles cordonnées(x2, y2) et le score),
- la performance de la communication via le DMA-Router. L'émission d'une seule donnée du N00 vers tous les nœuds de l'architecture nécessite 103 cycles d'horloges (T =4 x (diamètre=1) + 6 x (diamètre=2) + 4 x (diamètre=3) + 1 x (diamètre=4)),
- dans cet algorithme, et contrairement aux précédents algorithmes étudiés, le temps de traitement (complexité) est important par rapport au temps de communication.

5.5.4 Validation du squelette PIPE sur l'architecture proposée

Dans cette section, nous allons montrer la validation du squelette PIPE en implantant deux algorithmes de façon pipeliné. Le premier étage du pipeline implante l'algorithme de Harris et Stephen [78] qui permet la détection des points d'intérêts dans une image. Comme vu précédemment, le type de schéma de parallélisation SCM

est adapté à ce type d'algorithme (partage de données statique). Le deuxième étage du pipeline implante l'algorithme de mise en correspondance, ce dernier a été validé en mode FARM.
Dans une implantation pipelinée optimale, la difficulté revient à choisir correctement le nombre de processeurs affectés par étage. Cela doit être fait en tenant compte des précédents résultats et des contraintes temporelles de l'application finale.
Dans cet exemple, nous pouvons avoir différentes combinaisons. Afin d'illustrer la méthode et le fonctionnement de l'architecture, nous allons détailler la première proposition qui est 8 nœuds en SCM suivis de 8 nœuds en FARM, puis nous discutons les autres possibilités qui peuvent être envisagées dans le but de trouver la configuration optimale. La figure 5.12 montre le squelette PIPE tel qu'il est implanté dans l'architecture proposée.

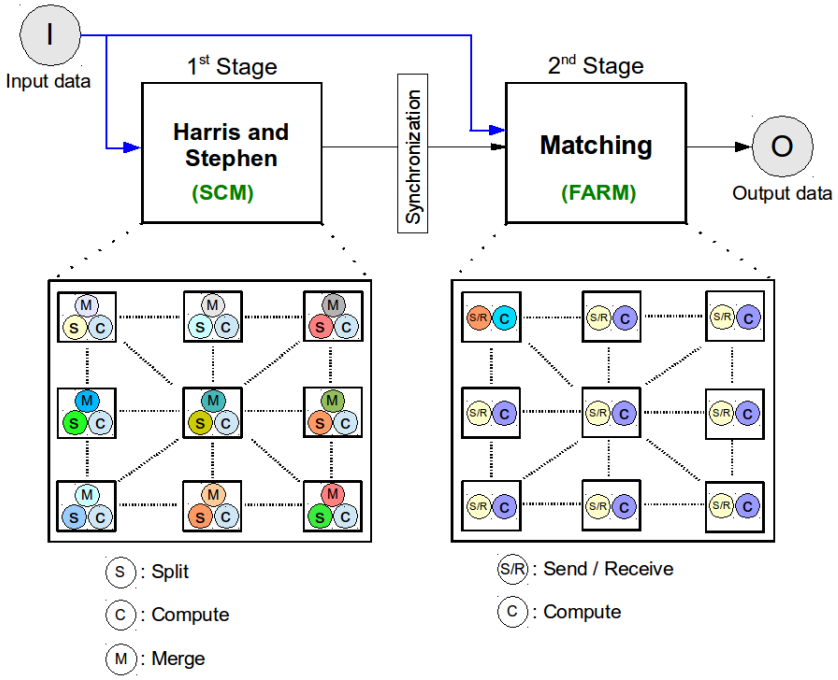


FIGURE 5.12 – Représentation graphique de la version implantée du squelette PIPE

5.5.4.1 Algorithme d'Harris et Stephen

L'algorithme de Harris et Stephen [78] permet la détection des points d'intérêts dans une image. Cet algorithme étudie la courbure des contours de l'image grâce à trois fonctions de convolution. Il permet soit de détecter des points "contours" soit des points "coins". Le coin d'un objet présent dans l'image est identifié lorsque les deux courbures principales sont élevées en un point. Le contour d'un objet est quant à lui identifié si seulement l'une des courbures est grande. La version utilisée ici comme

exemple provient des travaux de [81], [1] et [27]. La figure 5.13 présente un résultat de l'algorithme d'Harris et Stephen.

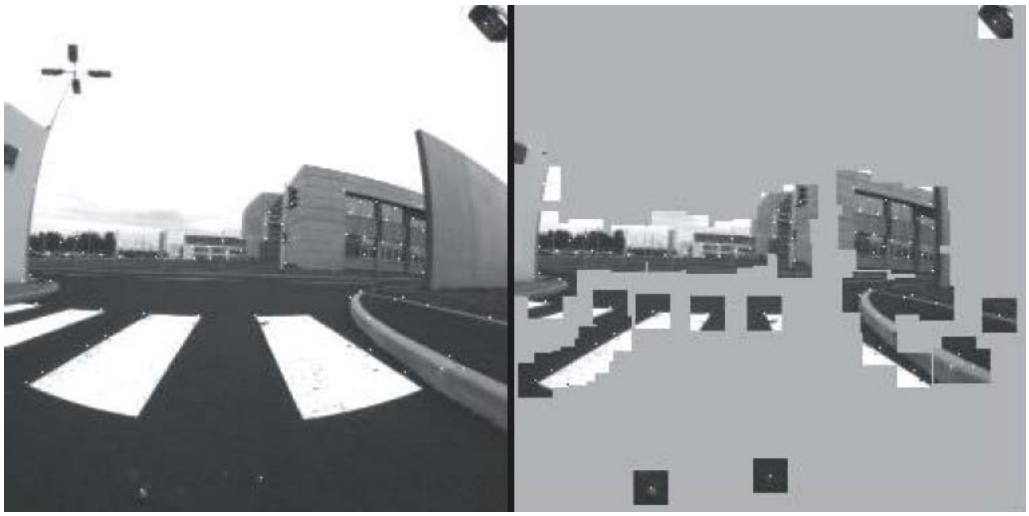


FIGURE 5.13 – Représentation graphique d'un résultat de l'algorithme d'Harris et Stephen

5.5.4.2 Résultats d'exécution

Le tableau 6.12 présente le résultat d'implantation de l'algorithme de Harris et Stephen sur l'architecture proposée. Les résultats sont présentés en temps d'exécution et accélération en fonction de la taille de l'image d'entrée et le nombre de nœuds utilisés dans l'implantation.

	Taille de l'image	**Implantations**				
		a (1)	**b (2)**	**c (4)**	**d (8)**	**e (16)**
Temps d'exécution (ms)	64x64	2,33	-	-	-	-
	256x256	37,36	18,73	9,41	4,75	2,43
	512x512	149,46	74,93	37,67	19,03	9,73
	1024x1024	597,87	299,75	150,69	76,15	38,95
	2048x2048	2391,49	1199,03	602,79	304,62	155,81
Accélération	64x64	1,00	-	-	-	-
	256x256	1,00	1,99	3,96	7,85	15,34
	512x512	1,00	1,99	3,96	7,85	15,34
	1024x1024	1,00	1,99	3,96	7,85	15,34
	2048x2048	1,00	1,99	3,96	7,85	15,34

TABLE 5.8 – Temps d'exécution et accélération de l'algorithme de Harris et Stephen sur l'architecture proposée

Le temps total d'exécution pour l'algorithme de Harris et Stephen peut être exprimé de la même façon que celui de l'algorithme du seuillage adaptatif, c'est-à-dire comme suit :

$$T_{total} = N_{envoi} \times (T_{compute} + T_{merge} + T_{transfer}) \tag{5.14}$$

Le tableau 6.12 montre une accélération proche de l'accélération idéale pour toutes les implantations. Ce résultat est dû aux mêmes raisons que vu précédemment. Toutefois, nous remarquons que ce résultat est moins bon que celui de l'algorithme de mise en correspondance même si la partie traitement est plus significative dans cet algorithme et la partie communication est moins importante. Ces résultats sont dûs au transfert de données qui se fait avec des liaisons point-à-point entre les nœuds, ce qui augmente la latence dans le cas de cet algorithme.

5.5.4.3 Implantation d'un pipeline de 8 nœuds en SCM suivis de 8 nœuds en FARM

L'idée pour la mise en œuvre d'un pipeline de deux étages (16 étages au maximum) est basée sur une utilisation optimale de la mémoire dans chaque étage. La taille de la sous-image à traiter (envoyée via le IFG) est cadencée par l'étage contenant le moins de nœuds donc le moins de mémoire globale pour l'étage. La figure 5.14 montre cette architecture pipelinée.

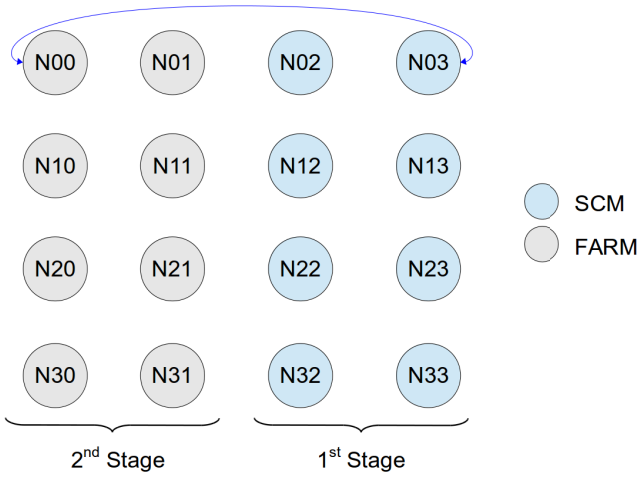


FIGURE 5.14 – Architecture a base de deux étages de pipeline

Dans cet exemple, nous traitons le cas de 8 nœuds en SCM suivis de 8 nœuds en FARM, solution la plus facile à mettre en œuvre. La taille de la sous-image injectée est de 8 x (64x64), ce qui fait une sous-image [9] de taille 128x256.

9. Pour rappel, la taille de la sous-image à traiter est de N_{noeuds} x S_{noeud} (i.e. 8 x (64x64)= 128x256),

L'algorithme de mise en correspondance utilise l'image courante et une image appelée image clé qui ici est l'image précédente. Afin de mettre en place ce contexte, trois banques mémoires sont nécessaires or un nœud ne dispose que de deux banques mémoires FM de 4ko chacune. Plusieurs solutions sont possibles, la plus simple à mettre en œuvre algorithmiquement étant d'utiliser les mémoires RM et SM, en sauvegardant une moitié dans la mémoire RM et l'autre moitié dans la mémoire SM (fig. 5.15). Effectivement, le deuxième étage est en mode FARM donc les deux mémoires sont peu utilisées. Après la sauvegarde de la demi-image, il reste 2 Ko libre dans RM et 2 Ko dans SM, ce qui est suffisant pour la transmission des données entre maître et esclaves pour cet algorithme (cordonnées). La figure 5.15 montre le contenu des mémoires FM dans le cas du première étage (à droite) et le contenu des mémoires FM, RM et SM dans le cas du deuxième étage (à gauche).

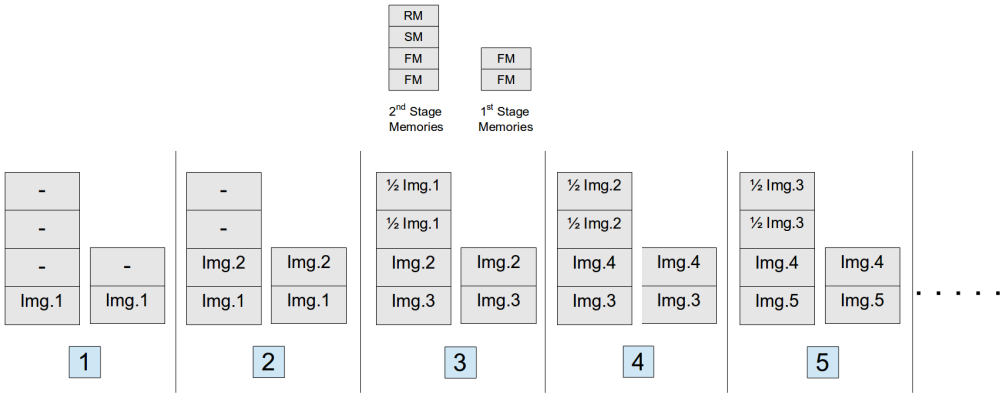


FIGURE 5.15 – Processus d'évolution de la mémoire sur les deux étages du pipeline en 8 nœuds en SCM, suivis de 8 nœuds en FARM

Processus d'exécution : Le processus d'exécution peut être expliqué comme suit :
1. Chargement de l'image 1 dans tous les nœuds.
2. Le premier étage commence le traitement sur l'image 1 (temps de traitement : 2,37 ms), tandis que le deuxième n'est pas sollicité.
3. Chargement de l'image 2 dans tous les nœuds.
4. Changement du mode de communication du N00 au FSL. Le premier étage envoi (par la voie du N03) les points d'intérêts de l'image 1 au N00 (maître).
5. Retour au mode de communication DMA-Routeur pour le N00. Le maître distribue les points d'intérêts aux esclaves. Tous les nœuds du deuxième étage copient [10] la moitié

ce qui implique que le traitement d'une image ($S_{globale}$) de 1024x1024 nécessite un $N_{envoi} = 32$ (voir la section 5.2.4).

10. La copie de l'image est transparente pour le premier étage, car ce dernier est en cours de traitement. A 400MHz, ce temps est négligeable (64x64 : 1024 cycles d'horloge ce qui fait $2,56\mu s$).

de l'image 1 dans la mémoire SM et l'autre moitié dans la mémoire RM, puis commence le traitement sur l'image 1.
6. Le deuxième étage à terminer le traitement (temps de traitement : 0,58 ms), le N00 récolte les résultats de mise en correspondance et transfert le résultat, mais le fait d'envoyer le résultat ne permet pas de passer à l'image suivante car l'image 2 est en cours de traitement par le premier étage.
7. Le premier étage a fini le traitement sur l'image 2.
8. re-bouclé à partir de 3 avec la nouvelle image.

Analyse : Le premier étage est l'étage le plus lent dans ce pipeline, ce qui fait un temps d'exécution total du pipeline égal au temps de cet étage soit 2,37 ms pour une image de 128x256 (donc 4,75 ms pour une image de 256x256). Ce résultat est intéressant car une exécution non pipelinée des deux algorithme (addition séquentielle des deux temps de traitement) donné un résultat de 5,92 ms (gain de 24,54%).

Toutefois, deux autres cas peuvent être envisagés :
- Cas 1 : il y a moins de points d'intérêt que de nœuds, dans ce cas le temps de traitement de l'étage est égale au moins au temps de traitement d'un point d'intérêt qui est toujours égale à environ 0,58 ms.
- Cas 2 : il y a plus de points d'intérêt que de nœuds, dans ce cas le temps de traitement dépend du nombre maximum de points d'intérêt dans un nœud de l'étage. Par exemple si on avait deux points d'intérêt dans un ou plusieurs nœuds, le temps de traitement du nœud doublera entraînant ainsi une augmentation (double) du temps de traitement de l'étage.

Selon l'algorithme à implanter, une solution est meilleure qu'une autre. La conclusion pour cet exemple dans le cas de ces deux algorithmes est que le premier étage et l'étage le plus lent, donc pour trouver l'implantation optimale, il faut augmenter le nombre de nœud de cet étage, sans toutefois pénaliser le deuxième étage dans le cas où on aura plus d'un point par nœud.

5.5.4.4 Nouvelle proposition d'implantation de pipeline de 4 nœuds en SCM suivis de 12 nœuds en FARM

En exploitant les résultats des implantations précédentes, une implantation pipelinée de 4 nœuds en SCM suivis de 12 nœuds en FARM semble mieux adaptée au prix du même effort de résolution du problème mémoire que vu précédemment (fig. 5.15). Dans cette implantation (fig. 5.16) deux cas sont possible vis-à-vis la taille de la sous-image à traiter :
- La taille de la sous-image à traiter est cadencée sur l'étage le plus petit. Dans ce cas, l'image d'entrée est de taille 64x256, 4 nœuds au premier étage pour effectuer le traitement peuvent suffire ce qui fait 2,35 ms pour traiter une image de 64x256 et donc 9,41 ms pour une image de 256x256.

Comparer à la première implantation, cette solution est deux fois plus lente. Ce résultat est bien sûr prévisible car le premier étage utilise que 4 nœuds (fig. 5.16), ce qui fait une architecture de 4 nœuds en SCM, suivis de 4 nœuds en FARM.

– Dans le cas où la sous-image est cadencée sur l'étage le plus grand, celle-ci doit être laissée plus longtemps sur le bus, pour que l'étage le plus petit arrive à faire le traitement en plusieurs fois. Le premier étage traite l'image de 192x256 en 2,42 ms ce qui est très intéressant.
Pour le deuxième étage, nous avons 12 points d'intérêts et seulement 4 nœuds pour faire le traitement, multipliant ainsi le temps de traitement par 3 ce qui donne environ 1,74 ms.
Le premier étage est toujours un peu plus lent que le deuxième ce qui donne un temps d'exécution total du pipeline égale a 2,42 ms pour une image de 192x256. Pour une image de 256x256 le temps de traitement est d'environ 3,22 ms au lieu de 4,75 ms trouver dans l'exemple précédent (un gain de 83,6% par rapport à la solution séquentielle).

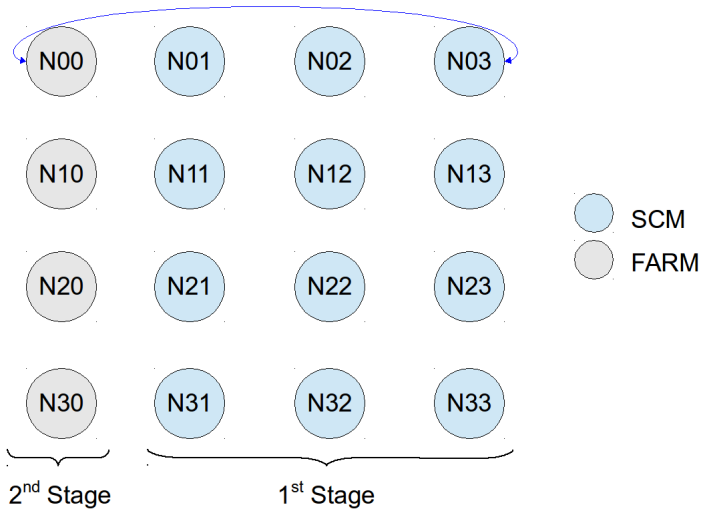


FIGURE 5.16 – Architecture a base de deux étage de pipeline en 12 nœuds en SCM, suivis de 4 nœuds en FARM

La difficulté pour la mise en œuvre de cette architecture est due au deuxième algorithme qui -utilisant l'image précédente- s'avère selon la mémoire disponible difficile à implanter. Toutefois, l'implantation proposée (8 nœuds en SCM suivis de 8 nœuds en FARM) est simple à mettre en œuvre et permet d'avoir un gain de 24,54% par rapport à l'exécution séquentielle dans le cas de ces deux algorithmes.

5.6 Conclusion

Dans ce chapitre, nous avons présenté le deuxième circuit réalisé dans cette thèse. L'architecture est constituée de 16 nœuds flexibles (de façon logicielle) par leur capacité de calcul, leur communication et leur gestion du flot vidéo. Cette approche s'appuie sur le paradigme des squelettes de parallélisation et les concepts AAA (Adéquation Architecture Algorithme) du traitement de l'image. Les performances (vitesse, consommation et surface) obtenus sont très intéressants et mettent clairement en avant le gain en passant du FPGA vers le ASIC. Les applications présentées illustrent les squelettes SCM, FARM et PIPE.

Les algorithmes mises en œuvre en utilisant le squelette SCM montrent des performances qui varient en fonction de la quantité de données à transmettre après la fin du traitement et la complexité du traitement en lui-même. Nous avons une accélération presque parfaite, dans le cas d'une grande complexité algorithmique et d'une faible communication (2 données par nœud avec l'algorithme de Harris et Stephen), puis une accélération moyenne dans le cas d'une très faible complexité algorithmique et d'une moyenne communication (256 données par nœud avec l'algorithme du calcul d'histogramme) et enfin une accélération faible, dans le cas d'une faible complexité algorithmique et d'une forte communication (1024 données par nœud) avec l'algorithme du seuillage adaptatif. Les résultats d'implantation utilisant le maximum de nœud donnent les meilleurs résultat en terme de temps d'exécution pour toutes les taille d'image sauf dans le cas de l'algorithme du calcul d'histogramme avec une image de 64x64.

L'implantation de l'algorithme de mise en correspondance en utilisant le squelette FARM a donné de très bons résultats sur toutes les implantations (accélération presque parfaite). Ce résultat est dû à la faible communication et la forte complexité de cet algorithme. Elle est due aussi au performance de la communication via DMA-Router.

Nous avons validé aussi une implantation pipelinée sur cette architecture en utilisant deux algorithmes (mise en correspondance et Harris et Stephen). Nous avons montré que dans le cas du pipeline la gestion de la mémoire est la partie critique et que certaines implantations ne peuvent pas être réalisées à cause de la non disponibilité de la mémoire. Nous avons montré aussi avec le premier exemple simple de pipeline un gain de 24,54% par rapport à l'exécution séquentielle. On rappelle que les tailles des mémoires ont été choisies pour rester réaliste en vue d'une réalisation d'ASIC (prix). La surface du circuit en ajoutant le seal ring est de $48,7mm^2$, ce qui fait un prix de 266000 euros pour la fabrication du circuit via le CMP.

Les résultats obtenus pour cette architecture sont très encourageants. Dans le chapitre suivant, nous allons présenter une amélioration de cette architecture sur plusieurs aspects.

Chapitre 6

HNCP-III : une architecture multiprocesseurs homogènes communicants à 64 cœurs en technologie ST 28nm FD-SOI

6.1 Introduction

Dans le chapitre précédant, nous avons présenté une architecture multiprocesseurs basée sur le concept des squelettes de parallélisation. La conception de cette architecture était basée sur des critères tels que : le choix de la topologie, la composition du nœud, la gestion du flot vidéo, les communications, etc....
Dans ce chapitre, nous présentons le troisième circuit développé dans le cadre de cette thèse. L'architecture est basée sur une extension du deuxième circuit (HNCP-II). Les améliorations consistent dans le passage de 16 nœuds à 64 nœuds (quatre clusters de 16 nœuds), l'utilisation d'une technologie plus intégrée (28 nm) avec un processus de fabrication différent (FD-SOI) et l'ajout d'un module pour la gestion de la consommation dans tous les nœuds (afin de réduire la consommation dynamique du circuit).

Le chapitre est organisé comme suit : la section 2 présente la technologie ST 28 nm FD-SOI. En section 3, nous décrivons l'implantation de l'architecture présentée dans le chapitre 5 sur cette technologie afin de voir le gain obtenu lors du passage de la 65 nm vers la 28 nm. L'architecture du troisième circuit est décrite en section 4. En section 5, nous proposons une solution pour le prototypage de l'architecture sur FPGA. Les résultats d'implantation en technologie ST 28 nm FD-SOI sont présentés en section 6. Une validation algorithmique de l'architecture est abordée en section 7. La section 8 conclut ce chapitre.

6.2 La technologie ST 28 nm FD-SOI

La technologie CMOS bulk est la technologie la plus utilisée depuis que le marché des ASIC existe. Comme les effets submicroniques deviennent de plus en plus prégnants, cette technologie a atteint certaines limites telles que les problèmes de mise à l'échelle, de courant de fuite et de variabilité du processus. Récemment, la technologie Fully Depleted Silicon on Insulator (FD-SOI) apparaît chez les fondeurs de circuits en s'appuyant sur une couche ultra mince de silicium sur un oxyde enfoui (Buried Oxide appelé BOx). Les transistors réalisés sur cette couche supérieure de silicium sont des dispositifs ultra minces (Ultra Thin Body) et présentent des caractéristiques intéressantes.
La technologie FD-SOI permet de résoudre avec moins de complexité du processus les problèmes se posant avec les technologies CMOS bulk "classique" en deçà de 28nm (voir la figure 6.1). En fait, l'approche FD-SOI offre un excellent contrôle électrostatique du transistor en améliorant ses performances et permet l'utilisation de tensions d'alimentation plus faibles (consommation d'énergie plus faible donc). En outre, la technologie FD-SOI réduit significativement les disparités de concentration des dopants réduisant ainsi drastiquement la variabilité de la tension de seuil des transistors. De plus, cette technologie propose intrinsèquement de faibles courants de fuite (avantage du SOI, silicium sur isolant) et permet une bonne maîtrise des effets de canal court (capacité à diminuer considérablement la longueur de la grille).

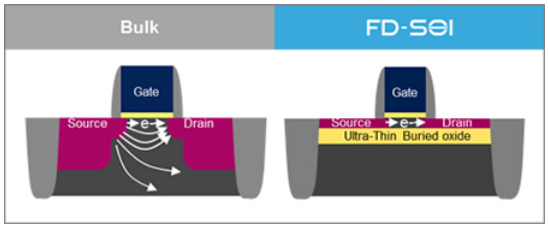


FIGURE 6.1 – La technologie Bulk Vs la technologie FD-SOI [69]

Le process CMOS 28nm de chez STMicroelectronics présente les caractéristiques suivantes :

- longueur du poly 28 nm,
- triple canal,
- fully depleted SOI devices,
- double transistors MOS Vt,
- double oxyde de grille (1.0V pour le core et 1.8V pour IO),
- double-damascène cuivre pour l'interconnexion,
- 10 couches de métal,
- pas de métallisation de 0,10 micron,
- support diverses alimentations : 1,8V, 1V,
- mémoire embarquée (simple port RAM /ROM / double port RAM),
- MIM (Metal-Insulator-Metal).

Le design-kit utilisé dans cette thèse (fournie par le CMP) se compose de 15 librairies de cellules standards (6 à base de 8 tracks [1] et 9 à base de 12 tracks) avec deux choix pour la tension de seuil : LL (LP Low Vt) et LR (LP Regular Vt) et un type PR (Place and Route) optimisé pour le placement routage. Pour ce qui est des librairies de cellules d'entrées/sorties (I/O), 8 bibliothèques sont disponibles. Les mémoires sont toujours sur commande et sans frais supplémentaire (hormis la surface en sus).

6.3 Résultats d'implantation du HNCP-II en technologie 28 nm FD-SOI

Dans cette section, nous allons présenter les résultats d'implantation du deuxième circuit sur la technologie ST 28 nm. Nous discutons aussi du gain obtenu en passant de la technologie 65 nm à la technologie 28 nm. Les résultats sont présentés en terme de fréquence, de surface, de puissance et d'estimations de temps d'exécution pour les différents algorithmes précédemment présentés.

6.3.1 Résultats après l'étape de synthèse logique

Les fréquences d'entrée de l'architecture sur la cible ASIC en utilisant la technologie 28 nm sont multipliés par 4. Nous avons $1,6$GHz pour le Core, le FPU, le "FPU Control Unit" et le DMA-Router et 800MHz pour le module "Frame Grabber", le bus Wishbone et les périphériques (GPIO, UART, INTC and Timer). La limite en fréquence de $1,6$GHz est aussi due à la conception du module FPU dont nous n'étions pas maître.

Pour ce qui est de la surface et de la consommation statique au **niveau nœud**, le tableau 6.1 présente les résultats de synthèse logique en terme de nombre de cellules, de surface et de consommation statique pour les composants constituant le nœud. La surface et la consommation statique totale d'un nœud avant placement-routage sont respectivement de $0,36mm^2$ et $47,83mW$.

Pour ce qui est de la surface et de la consommation statique au **niveau circuit**, le tableau 6.2 présente les résultats de synthèse logique en terme de nombre de cellules, de surface et de consommation statique. La surface et la consommation statique totale de cette architecture sont respectivement de $5,79mm^2$ et $769,46mW$, avant ajout des cellules d'entrées/sorties et avant placement-routage.

1. Un track est l'espace minimum toléré entre le métal 1 et le via du métal 1 dans une technologie donnée. Un track est généralement utilisé comme une unité pour définir la hauteur des cellules standard. Une cellule à 12 tracks est plus haute qu'une cellule à 8 tracks car plus d'espace de routage en metal 1 est disponible au sein de la cellule (ainsi les cellules seront plus rapides). Pour une cellule à 8 tracks, la cellule est plus compacte mais la vitesse est inférieure par rapport à une cellule à 12 tracks. Donc :
- 8 tracks : surface plus faible, moins de vitesse par rapport à la 12 tracks.
- 12 tracks : surface plus importante, plus de vitesse par rapport à 8 tracks.

Modules	**Nombre de cellules**	**Surface (um^2)**	**Consommation statique (mW)**
Memory Unit	263 (0,57%)	287526 (79,47%)	4,69 (9,80%)
Core	12946 (28,35%)	14925 (4,13%)	8,60 (17,98%)
DMA-Router	6446 (14,12%)	14042 (3,88%)	6,91 (14,45%)
FPU	8923 (19,54%)	10947 (3,03%)	7,15 (14,95%)
FPU Control Unit	547 (1,20%)	706 (0,19%)	0,40 (0,84%)
Frame Grabber	1201 (2,63%)	1818 (0,50%)	1,13 (2,36%)
JTAG	2312 (5,06%)	3684 (1,02%)	2,28 (4,77%)
Muxs, Buses,...	13028 (28,53%)	28165 (7,78%)	16,67 (34,85%)
Total	**45666 (100%)**	**361813 (100%)**	**47,83 (100%)**

TABLE 6.1 – Résultats d'implantation du nœud

Modules	**Nombre de cellules**	**Surface (μm^2)**	**Consommation statique (mW)**
16 Nodes	732224 (99,71%)	5792084 (99,95%)	767,69 (99,77%)
Wishbone bus	111 (0,01%)	95 (0,002%)	0,07 (0,01%)
TIMER	588 (0,08%)	895 (0,015%)	0,52 (0,07%)
UART	307 (0,04%)	384 (0,007%)	0,23 (0,03%)
INTC	191 (0,03%)	228 (0,004%)	0,13 (0,017%)
GPIO	65 (0,01%)	102 (0,002%)	0,06 (0,008%)
INPUT Frame Generator	415 (0,06%)	636 (0,01%)	0,36 (0,05%)
OUTPUT Frame Generator	413 (0,06%)	634 (0,01%)	0,36 (0,05%)
Clock,Resst,...	39 (0,005%)	60 (0,001%)	0,04 (0,005%)
Total	**734353 (100%)**	**5795118 (100%)**	**769,46 (100%)**

TABLE 6.2 – Résultats d'implantation du circuit

Le tableau 6.3 présente les résultats de consommation du circuit en fonction de la fréquence. L'analyse en puissance donnée par le synthétiseur (Design Compiler) montre une augmentation significative de la consommation statique (400% lors du passage de 200MHz à 800MHz et 200% lors du passage de 800MHz à 1600MHz). Pour la consommation interne des cellules, celle-ci présente une augmentation très faible (1,58% lors du passage de 200MHz à 800MHz et 2,88% lors du passage de 800MHz à 1600MHz).

Fréquences (MHz)	**consommation statique (mW)**	**consommation interne des cellules (mW)**
200	139,65	781,87
800	557,36	794,26
1600	1115	817,16

TABLE 6.3 – Résultats de consommation du circuit en fonction de la fréquence

6.3.2 Résultats après l'étape de placement-routage

La mémoire prend toujours la plus grande partie de la zone active du circuit (fig. 6.2). Le design-kit de la technologie 28 nm offre un large choix de mémoires (comparé à la technologie 65 nm). Les blocs mémoires utilisés sont de taille de 2Ko et 1Ko. La surface totale du circuit est de $22,13mm^2$ (deltaX=$4,892mm$ et deltaY=$4,523mm$). Le nombre d'entrées/sorties est évidement le même (326 incluant 48 plots d'alimentation). La fréquence estimée lors de la synthèse logique est maintenue ($1,6$GHz). L'analyse en puissance donne une consommation statique totale de $922,99mW$ et une consommation totale maximum de $16,58W$. La puissance dynamique a été calculée à l'aide de l'expression CV^2F en considérant le cas critique d'un taux d'activité égal à 1 (C : capacité extraite ($12,09nF$), V : tension d'alimentation ($0,9V$) et F : fréquence d'entrée ($1,6$GHz)) [2].

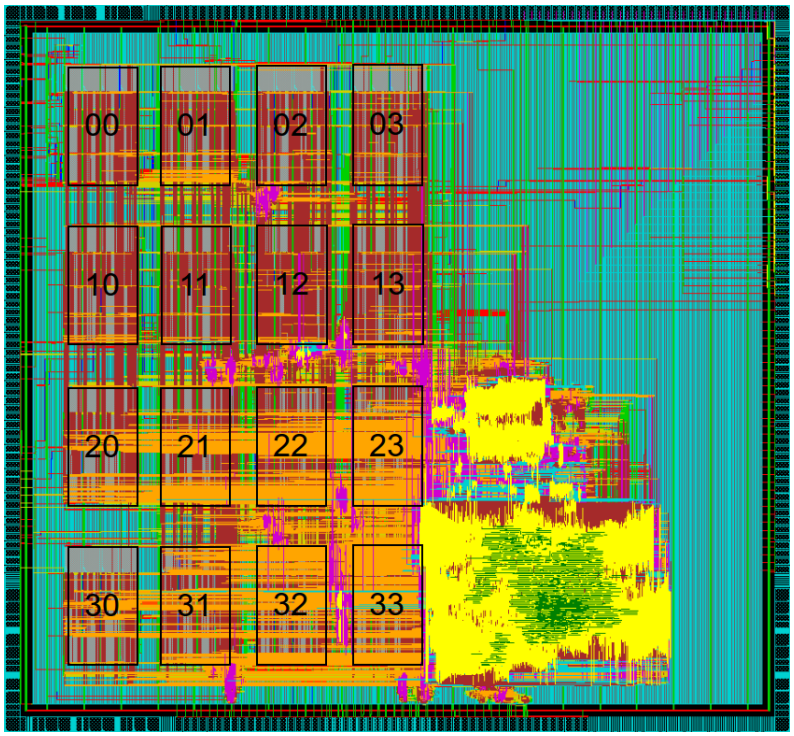


FIGURE 6.2 – Layout du circuit en 28 nm

Le tableau 6.4 montre la consommation statique du circuit en fonction des librairies utilisées. Contrairement au process 65 nm, les mémoires (Library DPHD_HIPERF_L et Library DPHD_HIGHPERF) ne consomment que 7,9%, alors que les cellules standard (C28SOI_SC_12_CORE_LL et C28SOI_SC_12_CLK_LL) consomment 92,05%.

2. Cette consommation élevée peut être expliquée par les mêmes raisons que vues précédemment.

Librairies	**Nombre de cellule**	**Consommation Statique (mW)**
DPHD_HIPERF_L	192 (0,01%)	52,44 (5,68%)
DPHD_HIGHPERF	128 (0,01%)	20,50 (2,22%)
C28SOI_SC_12_CORE_LL	838111 (47,79%)	788,88 (85,47%)
C28SOI_SC_12_CLK_LL	41342 (2,36%)	60,74 (6,58%)
C28SOI_SC_12_PR_LL	873254 (49,79%)	0,06 (0,01%)
C28SOI_IO_SF	278 (0,01%)	0,38 (0,04%)
C28SOI_IO_SF_BASIC	470 (0,03%)	0,002 (0,0002%)

TABLE 6.4 – Résultats de nombre de cellules et de consommation statique après placement routage par librairies

6.3.3 Estimations des temps d'exécution de l'HNCP-II en technologie ST 28 nm FD-SOI

A une fréquence de $1,6$GHz, les temps d'exécutions de tous les algorithmes présentés dans les tableaux du chapitre 5 sont divisés par 4. Le tableau 6.5 présente les résultats d'exécutions. Les accélérations sont toujours les mêmes (résultat attendu puisque c'est un rapport).

6.3.4 Comparaison de l'implantation en 65 nm et en 28 nm

Le tableau 6.6 montre la comparaison entre les performances obtenues avec la technologie 65 nm et la technologie 28 nm. Les résultats de surface et de fréquence sont conformes aux attentes. Pour la fréquence, un rapport de 4 (passage de 400MHz à $1,6$GHz) est observé de même que pour la surface (après synthèse la surface totale des cellules passe de $24,668mm^2$ à $5,795mm^2$). Pour ce qui est du layout, la surface est contrainte par le nombre d'entrées/sorties (IOs) : nous observons que la moitié du circuit est vide ce qui correspond bien au quart du layout en 65 nm. Par contres, les résultats de consommation ne sont pas conformes aux attentes. Ces résultats inattendus ont été envoyés au CMP (puis transmis à ST) qui ont confirmé la vraisemblance de ces résultats pour la technologie FD-SOI (le gain en consommation se faisant en dynamique). Nous rappelons que la surface à payer (tab.6.6) nécessite l'ajout du seal ring ($100\mu m^2$ de chaque cotés), ce qui fait un prix [3] de 266000 € en technologies 65 nm. En technologies 28 nm FD-SOI, le prix [4] et de 250000.

3. Pour un circuit de $49mm^2$ plus ou moins $5mm^2$ en 65nm CMOS (CMOS65LPGP) le CMP propose la formule suivante :
- $5mm^2$ premiers : 7500 €/mm^2 -> 37500 €
- $5mm^2$ à 15 mm^2 : 6000 €/mm^2 -> 60000 €
- $15mm^2$ à $54mm^2$: 5000 €/mm^2 -> 170000 €

4. Pour un circuit de $24mm^2$ en 28nm FDSOI (CMOS28FDSOI) le CMP nous a proposé un prix d'environ 250000 €. La formule donnée par le CMP [72] pour les circuits entre $5mm^2$ et 15 mm^2 donne un prix de 297600 €

	Taille de l'image	**Implantations**				
		a (1)	**b (2)**	**c (4)**	**d (8)**	**e (16)**
Temps d'exécution (ms) de l'algorithme du calcul d'histogramme	64x64	0,03	0,02	0,01	0,02	0,026
	256x256	0,53	0,28	0,15	0,09	0,05
	512x512	2,13	1,12	0,62	0,37	0,22
	1024x1024	8,52	4,50	2,49	1,48	0,88
	2048x2048	34,10	18,03	9,96	5,92	3,54
Temps d'exécution (ms) de l'algorithme du seuillage adaptatif	64x64	0,19	0,12	0,08	0,06	0,05
	256x256	3,16	1,98	1,38	1,08	0,92
	512x512	12,67	7,94	5,52	4,32	3,71
	1024x1024	50,69	31,77	22,11	17,28	14,86
	2048x2048	202,78	127,09	88,45	69,13	59,46
Temps d'exécution (ms) de l'algorithme de mise en correspondance	64x64	0,14	-	-	-	-
	256x256	2,32	1,16	0,58	0,29	0,14
	512x512	9,31	4,66	2,33	1,16	0,58
	1024x1024	37,25	18,64	9,32	4,67	2,34
	2048x2048	149,02	74,57	37,30	18,68	9,36
Temps d'exécution (ms) de l'algorithme de Harris et Stephen	64x64	0,58	-	-	-	-
	256x256	9,34	4,68	2,35	1,18	0,60
	512x512	37,36	18,73	9,41	4,75	2,43
	1024x1024	149,46	74,93	37,67	19,03	9,73
	2048x2048	597,87	299,75	150,69	76,15	38,95

TABLE 6.5 – Estimations des temps d'exécutions pour les quatre algorithmes sur l'architecture du deuxième circuit en 28 nm

Technologies	**CMOS 65 nm**	**FD-SOI 28 nm**
Fréquence (MHz)	400	1600
Consommation Statique (mW)	30,44	769,46
Surface Cellules / Layout (mm^2)	24,67 / 45,89	5,79 / 22,13
Surface à payer (mm^2)	48,7	24,05
Coût (€)	266000	250000

TABLE 6.6 – Comparaison de performance et de prix entre la 65 nm et la 28 nm

Il est évident que le passage de la technologie 65nm à la technologie 28nm permet d'améliorer les performances de l'HNCP-II. Cette amélioration est confirmée pour la fréquence (de 400MHz à $1,6$GHz), moyennement intéressante pour la surface (de $45{,}89mm^2$ à $22{,}13mm^2$) à cause du nombre d'IOs et à revoir en terme de consommation statique ($30{,}44mW$ à $769{,}46mW$).
Dans la section suivante, nous présentons l'architecture du HNCP-III dans la quelle l'aspect technologique de la 28 nm est gardé et l'aspect architectural est amélioré notamment pour ce qui est du nombre de cœurs.

6.4 Architecture proposée du HNCP-III

Dans cette section, nous allons tout d'abord donner une vue d'ensemble de l'architecture globale, puis au niveau cluster et enfin au niveau nœud. Nous discutons également du domaine d'horloge et de l'ajout du module de gestion de consommation.

6.4.1 Architecture du circuit

Le circuit est constitué de quatre clusters (regroupement de 16 nœuds). Un seul module "Input Frame Generator" est implanté considérant que nous avons une seule source d'image. Le signal vidéo est simple à gérer ayant une fréquence de fonctionnement inférieure à celui des cœurs de processeur. Les clusters sont connectés entre eux via des liens directs (Asynchrone FIFO) comme le montre la figure 6.3.
Dans cette architecture, nous avons supprimé le module "Output Frame Generator", car nous considérons que les résultats (images, points d'intérêts, cordonnés, vecteurs,...) peuvent être gérés plus efficacement à l'extérieur du circuit (par un "simple" FPGA par exemple).

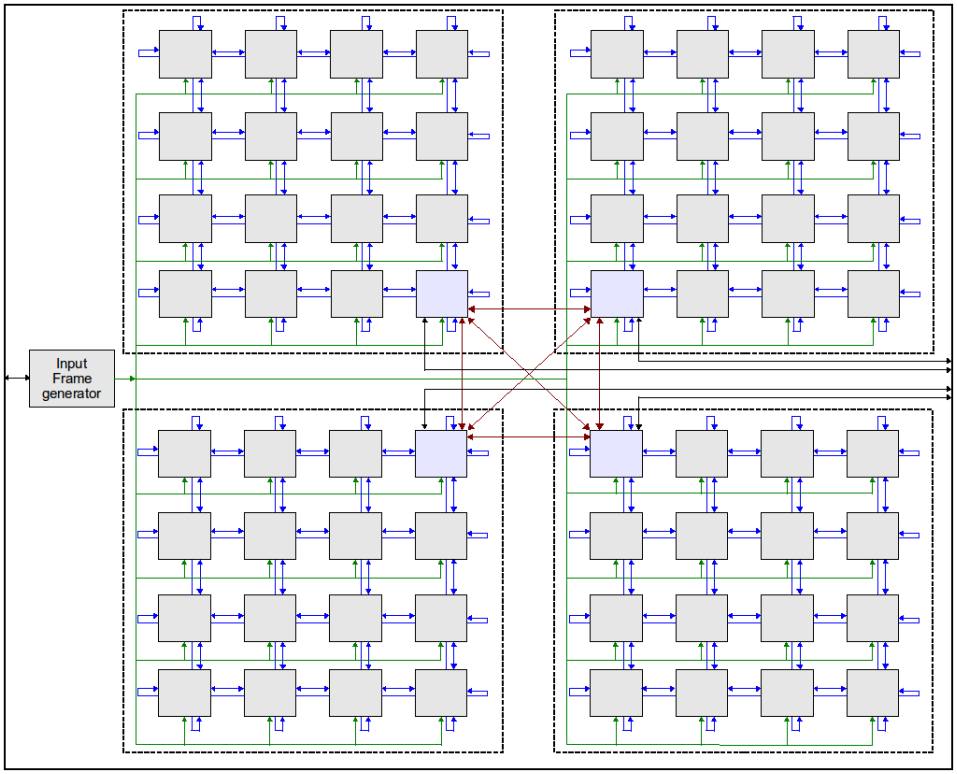


FIGURE 6.3 – Architecture du circuit HNCP-III

6.4.2 Architecture du cluster

L'architecture d'un cluster correspond à l'architecture du deuxième circuit présenté dans le chapitre 5 avec le nœud 00 qui a accès aux périphériques via le bus Wishbone (communication de flots de données sortants du cluster). Le fait d'avoir des clusters permet :

- Une programmation (aspect logiciel) de la communication plus simple.
- Une étape de fusion plus simple grâce à la disponibilité de la mémoire, dans le cas d'une communication dense (algorithme de seuillage adaptatif).
- Une réalisation plus simple au niveau physique (les nœuds de bordure d'un côté sont connectés aux nœuds de bordure du côté opposé).
- Une gestion d'horloge locale.
- Une testabilité modulaire (chaque cluster a sa propre chaîne JTAG).

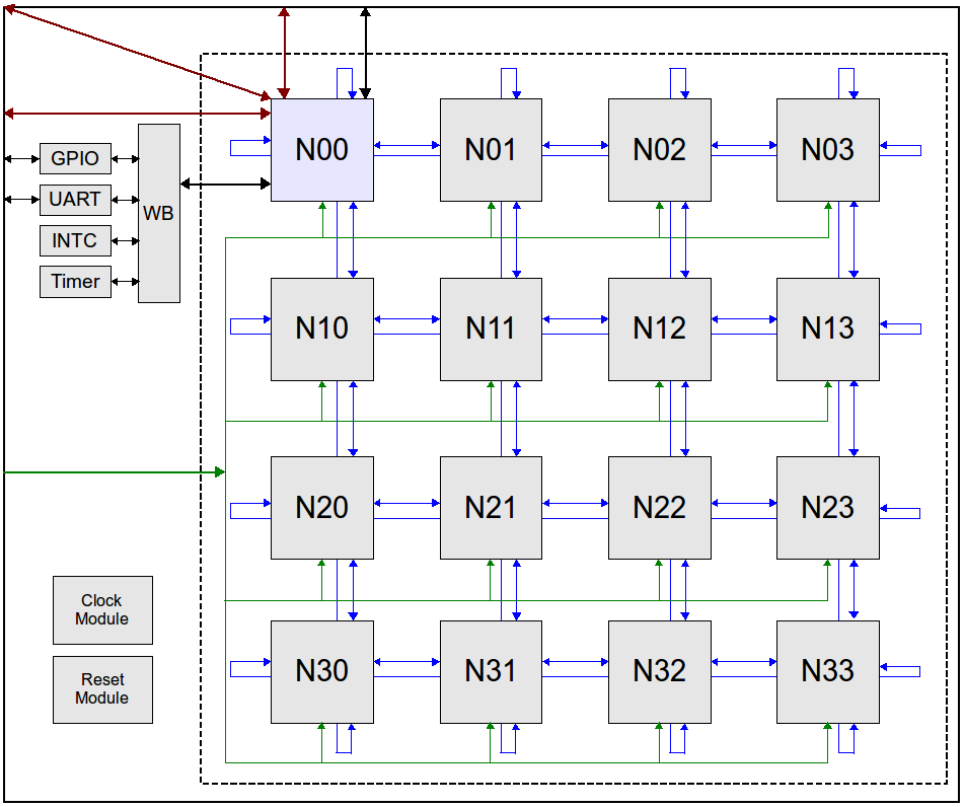


FIGURE 6.4 – Architecture d'un cluster HNCP-III

6.4.3 Architecture du nœud

L'architecture d'un nœud est proposée en figure 6.5. Par rapport au deuxième circuit (HNCP-II), un module "power management" est ajouté pour la gestion de la consommation. Ce dernier grâce à une instruction FSL peut couper l'horloge de n'importe quel module dans le nœud y compris le core (c'est-à-dire qu'on peut mettre tous les modules du nœud en veille). Cette méthode est appelée "clock gating" et consiste à couper le signal d'horloge d'une partie du circuit lorsque celle-ci est inactive.

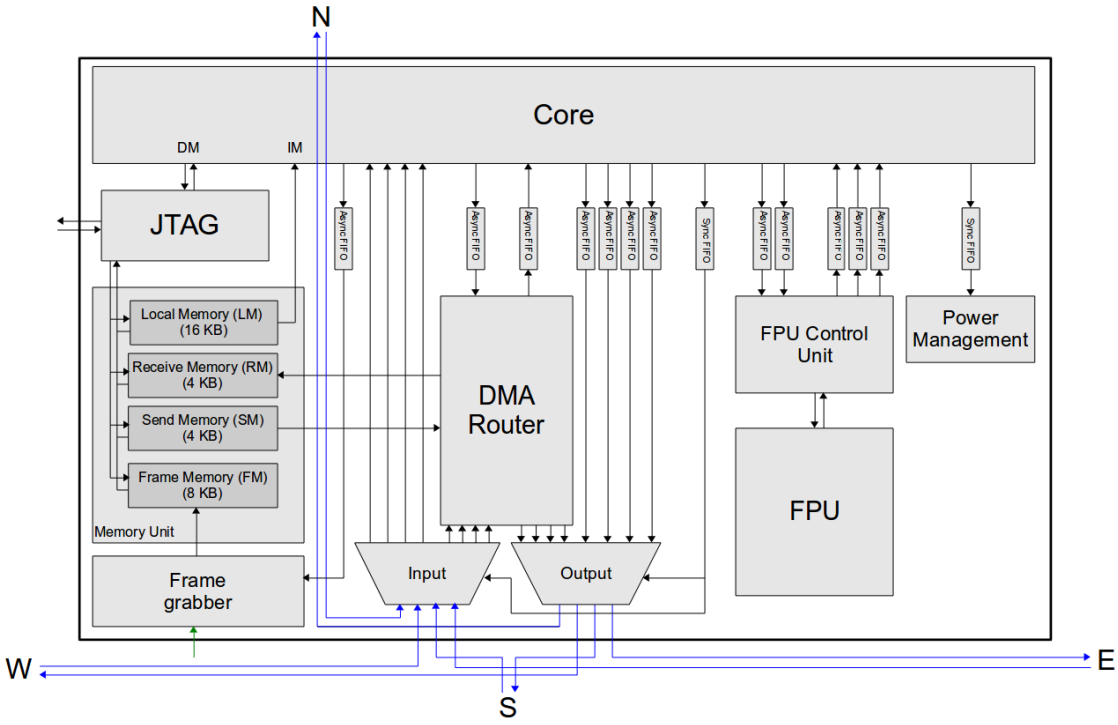


FIGURE 6.5 – Architecture d'un nœud HNCP-III

6.4.4 Domaine d'horloge

Comme le montre la figure 6.6, les modules de la même couleur fonctionnent à la même fréquence. L'arbre d'horloge d'un circuit synchrone représente généralement une part non négligeable de sa consommation dynamique.
La technique du "clock gating" n'a aucune influence sur la consommation statique.

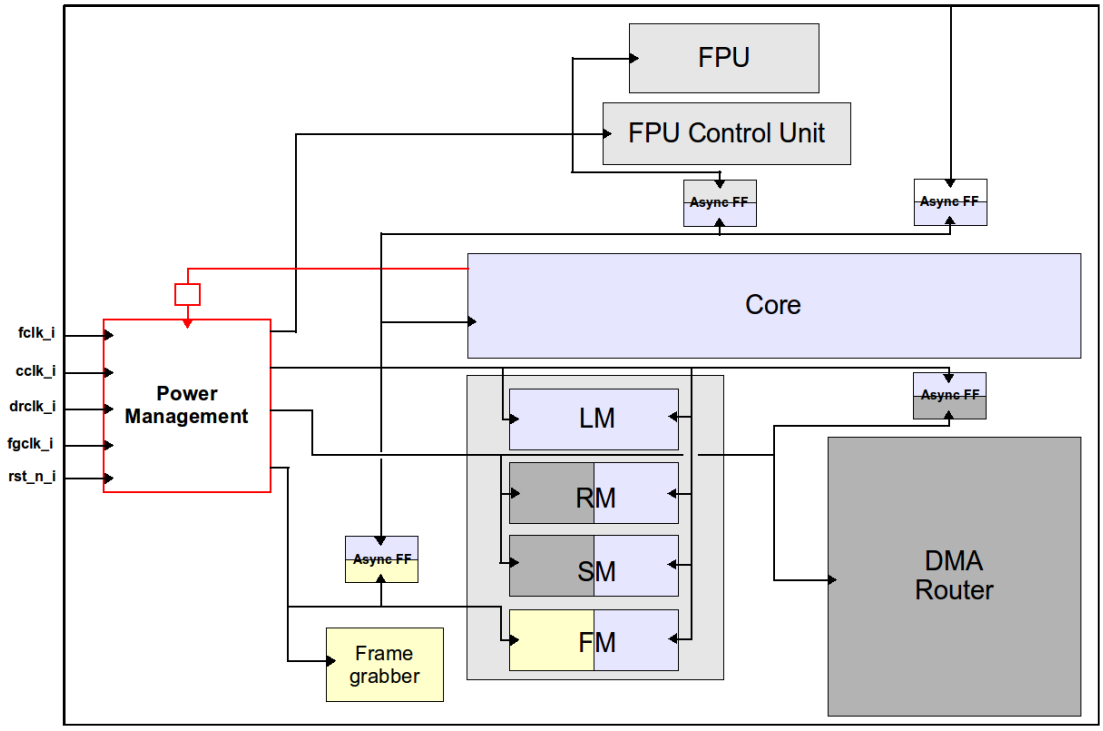


FIGURE 6.6 – Domaine d'horloge

6.5 Problématique du prototypage sur FPGA

Le plus grand FPGA de chez Xilinx est le Virtex-7 [5] XC7V2000T. L'architecture proposée est environ 4 fois plus grande que l'architecture du deuxième circuit, ce qui fait une estimation de 371596 slices. Or le Virtex-7 XC7V2000T ne contient que 305400 Slices.

Le prototypage du troisième circuit nécessiterait l'utilisation des système multi-FPGA tels que ceux proposé par proFPGA [82] pour synthétiser l'architecture proposée (par exemple le "duo V7" qui est constitué de deux FPGA Virtex-7 XC7V2000T ou encore le "quad V7" qui est constitué de quatre FPGA Virtex-7 XC7V2000T).

Ce prototypage n'a pas été mis en œuvre à cause de la non disponibilité d'une carte d'évaluation de ce type. Nous notons également que ce type de FPGA nécessite une licence spécifique. La solution ci-dessus est une proposition pour le prototypage du HNCP-III.

5. La décomposition d'un Slice du Virtex-6 est la même que celui du Virtex-7, c'est-à-dire quatre LUTs et huit flip-flops

6.6 Résultats d'implantation du HNCP-III en technologie ST 28 nm FD-SOI

Dans cette section, nous allons présenter les résultats d'implantations du troisième circuit sur la technologie ST 28 nm FD-SOI. Les résultats sont présentés en terme de fréquence, de surface et de puissance.

6.6.1 Résultats après l'étape de synthèse logique

Les fréquences d'entrées de l'architecture sont de $1,6$GHz pour le core, le FPU, le "FPU Control Unit" et le DMA-Router, 800MHz pour le module "Frame Grabber", le bus Wishbone et les périphériques (GPIO, UART, INTC and Timer).

Le tableau 6.7 présente les résultats de synthèse logique en terme de nombres de cellules, de surface et de consommation statique pour l'architecture présentée sur la figure 6.3. La surface et la consommation statique totale de cette architecture sont respectivement de $23,31mm^2$ et $3,14W$, avant ajout des cellules d'entrées/sorties et avant placement-routage. Le nombre de cellules et la surface du nœud ont augmenté atteignant respectivement 46476 cellules et 363627 μm^2 contre 45666 cellules et 361813 μm^2 dans la précédente architecture. La raison de cette augmentation est la mise à jour du DMA-Router (il est passé à 7227 cellules (pour 15787 μm^2) au lieu de 6446 (pour 14042 μm^2) dans la précédente architecture. L'ajout du module de gestion d'horloge (qui implique aussi l'ajout d'une FIFO synchrone) a fait augmenter la surface du nœud de 29 cellules (pour 69 μm^2).

Modules	**Nombre de cellules**	**Surface (μm^2)**	**Consommation statique (mW)**
Node	46476	363627	48,73
Cluster	749792	5829787	787,45
Total	**2999564**	**23319779**	**3147,12**

TABLE 6.7 – Résultats d'implantation du circuit

Le tableau 6.8 présente les résultats de consommation du circuit en fonction de la fréquence. L'analyse en puissance donnée par le synthétiseur (Design Compiler) montre une augmentation significative de la consommation statique. Pour la consommation interne des cellules, celle-ci présente une augmentation très faible.

Fréquences (MHz)	**consommation statique (W)**	**consommation interne des cellules (W)**
200	0,39	3,20
800	1,54	3,26
1600	3,08	3,36

TABLE 6.8 – Résultats de consommation du circuit en fonction de la fréquence

6.6.2 Résultats après l'étape de placement-routage

La mémoire prend toujours la plus grande partie de la zone active du circuit (vue Amoeba sur la figure 6.7). La surface totale du circuit est de $48,30mm^2$ (deltaX = $5,431mm$ et deltaY = $8,894mm$). Le nombre d'entrées/sorties est de 497 (incluant 96 plots d'alimentation). L'analyse en puissance donne une consommation statique totale de $3,14W$ et une consommation totale maximum de $98,48W$. La puissance dynamique a été calculée à l'aide de l'expression CV^2F en considérant le cas critique d'un taux d'activité égal à 1 (C : capacité extraite ($73,57nF$), V : tension d'alimentation ($0,9V$) et F : fréquence d'entrée ($1,6$GHz)) [6]. Comme dit précédemment, la consommation statique avec la technologie FD-SOI est très élevée. La technique du "clock gating" n'a aucune incidence sur la consommation statique du circuit. Pour réduire cette dernière, il faut réduire ou couper totalement l'alimentation d'un sous-circuit. Nous parlons dans ce cas de "power gating".

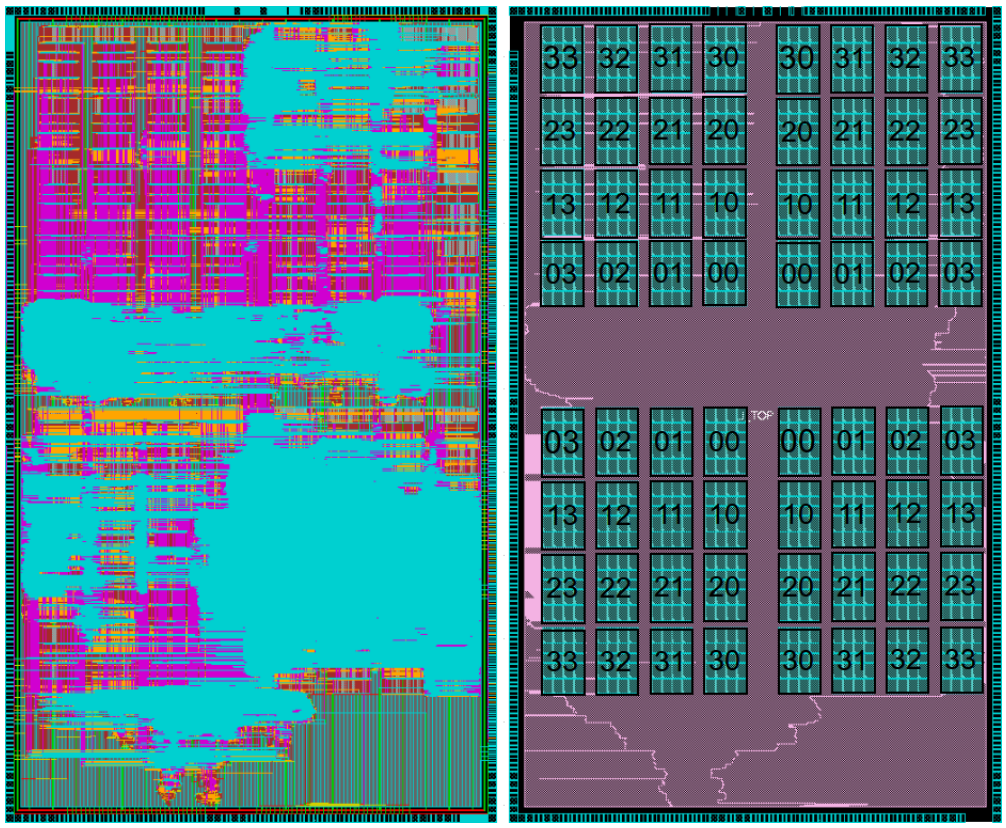


FIGURE 6.7 – Layout du circuit vue physique (à gauche) vue Amoeba (à droite)

6. Cette consommation élevée peut être expliquée par les mêmes raisons que vues précédemment.

6.7 Validations algorithmiques

Dans cette section, nous présentons la validation de l'architecture proposée sur les trois squelettes de parallélisation supportés par celle-ci en utilisant les mêmes algorithmes que précédemment.

Les résultats des temps d'exécution présentés dans ce chapitre sont calculés à partir des résultats des temps d'exécution trouvés dans le chapitre 5 et non pas simulés (comme pour l'HNCP-II) à cause de la taille de l'architecture (blocage du simulateur). Ceci lève un point important concernant les outils matériels utilisés, qui ne sont pas adaptés à des conceptions aussi importantes [7]. Toutefois, nous rappelons que les simulations effectuées dans le chapitre 5 sont avec une précision au coup d'horloge [8], ce qui permet d'avoir une estimation très précise pour les temps d'exécutions (puis les accélérations) de l'HNCP-III.

6.7.1 Configurations architecturales d'évaluations

La figure 6.8 présente trois configurations d'implantations algorithmiques dans l'architecture proposée. Dans (f) l'algorithme est implanté sur 32 nœuds, dans (g) l'algorithme est implanté sur 48 nœuds et enfin dans (h) l'algorithme est implanté sur tous les nœuds. Nous rappelons que le cas de 16 nœuds représente l'implantation d'un cluster.

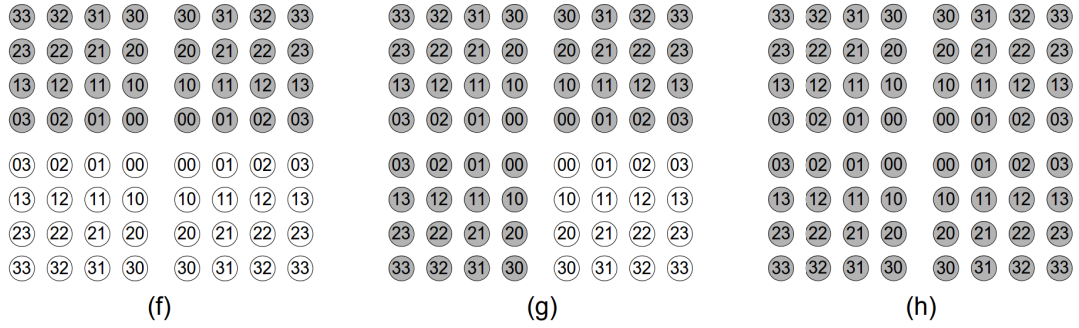


FIGURE 6.8 – Configurations architecturales d'évaluations

7. Ce qui augmente considérablement les temps de conception. Le temps nécessaire (selon les contraintes) pour effectuer les étapes de la synthèse logique et du placement-routage pour cette architecture est de plusieurs jours.

8. Les temps d'exécutions et les accélérations sont présentés sur tous les tableaux avec deux chiffres après la virgule, ceci a été effectué après la fin des calcules pour simplifier la lecture.

6.7.2 Validation du squelette SCM

6.7.2.1 Résultats d'exécution de l'algorithme du calcul d'histogramme

Une fois qu'un cluster a terminé son traitement, il envoie son résultat au cluster 00 (choix initial, un autre cluster aurait pu être choisi). Ce dernier effectue l'addition de son résultat avec les résultats reçus, puis envoie le résultat final à l'extérieur. Le temps total d'exécution peut être exprimé comme suit :

$$T_{cluster} = (T_{compute} + T_{merge}) + T_{send\_to\_cluster00} = 0,05ms \tag{6.1}$$

$$T_{cluster00} = (T_{compute} + T_{merge}) + T_{add} + T_{send\_to\_out} = 0,05ms \tag{6.2}$$

$$T_{total} = N_{envoi} \times (T_{cluster} + (N_{envoi} - 1) \times T_{add}) + T_{send\_to\_out} \tag{6.3}$$

Sachons que T_{add}=1,6μs et que $T_{send\_to\_out}$=1,76μs. Le T_{total} peut être estimé au :

$$T_{total} = N_{envoi} \times T_{cluster} \tag{6.4}$$

Le tableau 6.9 présente le résultat d'exécution de l'algorithme du calcul d'histogramme sur l'architecture proposée. Les résultats sont présentés en temps d'exécution et accélération en fonction de la taille de l'image d'entrée et du nombre de nœuds utilisés dans l'implantation. Afin de calculer l'accélération des différentes implantations, les temps d'exécutions d'un seul nœud (implantation (a) dans le chapitre 5) avec les différentes tailles d'images sont :
$T_{1(512x512)}$ = 2,13 ms
$T_{1(1024x1024)}$ = 8,52 ms
$T_{1(2048x2048)}$ = 34,10 ms

	Taille de l'image	**Implantations**		
		f (32)	**g (48)**	**h (64)**
Temps d'exécution (ms)	512x512	0,11	0,11	0,05
	1024x1024	0,44	0,33	0,22
	2048x2048	1,77	1,22	0,88
Accélération	512x512	19,18	19,18	38,36
	1024x1024	19,17	25,57	38,35
	2048x2048	19,17	27,89	38,35

TABLE 6.9 – Temps d'exécution et accélération de l'algorithme du calcul d'histogramme sur l'architecture HNCP-III

Les temps d'exécutions obtenus pour cet algorithme sont très intéressants. Un traitement en temps réel est largement assuré par cette architecture pour les trois implantations. L'accélération idéale est obtenu lorsque S_n = n (Donc 32, 48 et 64 pour (f), (g) et (h) successivement). Nous remarquons que l'accélération augmente dans l'implantation (g) : cela est dû au fait que dans le cas d'une image 512x512, les 3 quarts de l'image sont traités en une seule fois puis le quatrième quart est traité par un seul

cluster de l'architecture. Nous obtenons donc le même résultat que pour (f) (c'est-à-dire traitement en deux fois). Dans le cas 1024x1024, 5 envois sont traités par les trois clusters et le dernier envoi est traité par un cluster. Enfin dans le cas 2048x2048, 21 envois sont traités par les trois cluster et le dernier envoi est traité par un cluster.

6.7.2.2 Résultats d'exécution de l'algorithme du seuillage adaptatif

Le tableau 6.10 présente le résultat d'implantation de l'algorithme du seuillage adaptatif sur l'architecture proposée. Les résultats sont présentés en temps d'exécution et accélération en fonction de la taille de l'image d'entrée et du nombre de nœuds utilisés dans l'implantation. Afin de calculer l'accélération des différentes implantations, les temps d'exécutions d'un seul nœud (implantation (a) dans le chapitre 5) avec les différentes tailles d'images sont :
$T_{1(512x512)}$ = 12,67 ms
$T_{1(1024x1024)}$ = 50,69 ms
$T_{1(2048x2048)}$ = 202,78 ms

	Taille de l'image	**Implantations**		
		f (32)	**g (48)**	**h (64)**
Temps d'exécution (ms)	512x512	1,85	1,85	0,92
	1024x1024	7,43	5,57	3,71
	2048x2048	29,73	20,44	14,86
Accélération	512x512	6,81	6,81	13,63
	1024x1024	6,81	9,09	13,63
	2048x2048	6,81	9,91	13,63

TABLE 6.10 – Temps d'exécution et accélération de l'algorithme du seuillage adaptatif sur l'architecture HNCP-III

Les temps d'exécutions obtenus permettent toujours d'avoir un traitement en temps réel avec cette architecture pour les trois implantations. Le tableau 6.10 montre une plus faible accélération que celle de l'algorithme du calcul d'histogramme. Ce résultat est dû au temps de fusion important (dû à la quantité de donnée à transférer).

6.7.3 Validation du squelette FARM

Dans toutes les implantations effectuées dans cette section, le nœud N00 est considéré comme le maître et les autres sont considérés comme des esclaves pour chaque cluster.

6.7.3.1 Résultats d'exécution de l'algorithme de mise en correspondance de primitives

Le tableau 6.11 présente le résultat d'implantation de l'algorithme de mise en correspondance sur l'architecture proposée. Les résultats sont présentés en temps d'exécution

et accélération en fonction de la taille de l'image d'entrée et du nombre de nœuds utilisés dans l'implantation. Afin de calculer l'accélération des différentes implantations, les temps d'exécutions d'un seul nœud (implantation (a) dans le chapitre 5) avec les différentes tailles d'images sont :
$T_{1(512x512)}$ = 9,31 ms
$T_{1(1024x1024)}$ = 37,25 ms
$T_{1(2048x2048)}$ = 149,02 ms

	Taille de l'image	**Implantations**		
		f (32)	**g (48)**	**h (64)**
Temps d'exécution (ms)	512x512	0,29	0,29	0,14
	1024x1024	1,17	0,87	0,58
	2048x2048	4,68	3,22	2,34
Accélération	512x512	31,81	31,81	63,62
	1024x1024	31,81	42,41	63,62
	2048x2048	31,81	46,27	63,62

TABLE 6.11 – Temps d'exécution et accélération de l'algorithme de mise en correspondance sur l'architecture HNCP-III

Le tableau 6.11 montre des résultats de temps d'exécution très intéressants pour cet algorithme. Le tableau 6.11 montre une accélération proche de l'accélération idéale pour toutes les implantations.

6.7.4 Validation du squelette PIPE

Dans cette section, nous allons montrer la validation du squelette PIPE sur l'architecture proposée en implantant deux algorithmes (mise en correspondance et Harris et Stephen) utilisés précédemment de façon pipeliné. Mais avant la mise en œuvre de cette validation, nous présentons le résultat d'implantation de l'algorithme de Harris et Stephen.

6.7.4.1 Résultats d'exécution de l'algorithme de Harris et Stephen

Le tableau 6.12 présente le résultat d'implantation de l'algorithme de Harris et Stephen sur l'architecture proposée. Les résultats sont présentés en terme de temps d'exécution et d'accélération en fonction de la taille de l'image d'entrée et du nombre de nœuds utilisés dans l'implantation. Les temps d'exécutions d'un seul nœud (implantation (a) dans le chapitre 5) avec les différentes tailles d'images sont :
$T_{1(512x512)}$ = 37,36 ms
$T_{1(1024x1024)}$ = 149,46 ms
$T_{1(2048x2048)}$ = 597,87 ms

	Taille de l'image	Implantations		
		f (32)	g (48)	h (64)
Temps d'exécution (ms)	512x512	1,21	1,21	0,60
	1024x1024	4,86	3,65	2,43
	2048x2048	19,47	13,39	9,73
Accélération	512x512	30,69	30,69	61,39
	1024x1024	30,69	40,92	61,39
	2048x2048	30,69	44,65	61,39

TABLE 6.12 – Temps d'exécution et accélération de l'algorithme de Harris et Stephen sur l'architecture HNCP-III

Contrairement au deuxième circuit (HNCP-II), toutes les implantations arrivent à traiter les images de taille 1024x1024 et 2048x2048 en temps réels (pour un capteur standard (40 ms)). Le tableau 6.12 montre une accélération proche de l'accélération idéale pour toutes les implantations. Ce résultat est dû aux mêmes raisons que celles expliquées précédemment.

6.7.4.2 Exemple de pipeline : 32 nœuds en SCM suivis de 32 nœuds en FARM

La figure 6.9 montre une extension de l'architecture pipelinée (32 nœuds en SCM suivis de 32 nœuds en FARM) mis en œuvre dans le chapitre 5.

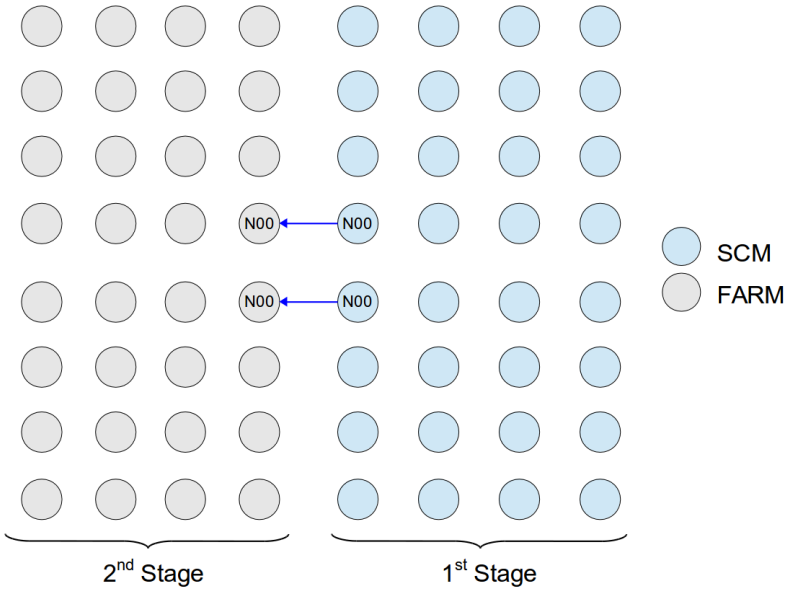


FIGURE 6.9 – Architecture a base de deux étage de pipeline

Pour une image de 512x512, le temps d'exécution du premier étage est de 1,21 ms et celui du deuxième étage est de 0,29 ms. Donc, l'étage 1 est l'étage le plus lent dans ce pipeline ce qui donne un temps d'exécution total égal à 1,21 ms.
Ce résultat représente un gain de 20% par rapport au 1,51 ms dans le cas d'une exécution non pipelinée (séquentielle). Nous pouvons également noter que ce résultat est environ 15 fois plus rapide que celui de l'HNCP-II (19,03 ms).

6.8 Conclusion

Dans ce chapitre, nous avons présenté le troisième circuit réalisé dans le cadre de cette thèse. L'architecture est constituée de 4 clusters de 16 nœuds. Les performances obtenues en utilisant la technologie 28 nm FD-SOI sont très intéressantes sauf pour la consommation statique du circuit ce qui nous amène à mettre en œuvre la technique du "power gating" qui permet de réduire la consommation statique. Les algorithmes présentés illustrent les squelettes SCM, FARM et PIPE. Les résultats d'implantations utilisant le maximum de nœud donnent les meilleurs résultat en terme de temps d'exécution pour toutes les tailles d'image et permettent, largement, d'assurer un traitement en temps réel du système sur l'ensemble des applications visées.

Conclusion et perspectives

Le Réseau Homogène de Processeurs Communicants a montré son efficacité dans les applications de traitements d'images par les précédents travaux de thèses [25] [1] [27]. Tous ces travaux sont basés sur une cible FPGA permettant ainsi, grâce à la flexibilité de ces composants, de mettre en place une méthodologie de prototypage rapide. Cette méthodologie combine l'aspect matériel basé sur une architecture multiprocesseurs homogène communicants et l'aspect logiciel basé sur les squelettes de parallélisation.

Ces travaux de thèse s'inscrivent dans un contexte de continuité avec une cible technologique différente : la cible ASIC. Ce changement de cible a finalement affecté profondément l'aspect matériel de cette méthodologie, nous amenant à re-créer le nœud de base constituant l'architecture multiprocesseurs notamment le cœur du processeur. Après des évaluations effectuées sur des processeurs libres, le choix du SecretBlaze s'est imposé. Toutefois et afin d'intégrer ce processeur dans la méthodologie, des améliorations et des adaptations ont été apportées en ajoutant des instructions de type FSL (pour avoir les mêmes fonctionnalités que le MicroBlaze) et un système FPU (pour avoir un calcul matériel en virgule flottante performant). Un système de gestion de trame permet à un nœud l'accès direct au flot vidéo. Enfin la communication entre nœuds du réseau est basée sur deux modes : le premier utilise des liens point-à-point assurant la mise en œuvre du squelette SCM et le deuxième utilise le DMA-Routeur assurant la mise en œuvre du squelette FARM. Des améliorations ont été apportées (ajout de nouvelles topologies et ré-usinage) au DMA-Routeur. Un module JTAG a également été ajouté afin de permettre la programmation et le debug du système.

Parmi les résultats importants de cette thèse, nous pouvons indiquer la fabrication du premier circuit numérique dans la technologie ST 65nm à l'Institut Pascal. Ce circuit doit ouvrir le laboratoire à la fabrication de circuits intégrés numériques plus complexes dans l'avenir.

Les deux architectures multiprocesseurs (16 cœurs en 65nm CMOS et 64 cœurs en 28nm FD-SOI) proposées dans le cadre de cette thèse montrent les performances élevées que nous pouvons obtenir sur cible ASIC. En s'inspirant de l'aspect configurable des FPGA, nous avons proposé -dans les limites imposées par la cible ASIC- des architectures flexibles portées par des nouveautés architecturales. Nous avons pu également

valider les trois squelettes de parallélisation (SCM, FARM et PIPE).

Un autre résultat significatif important de ces travaux est la contribution dans le domaine pédagogique, inspirée des contraintes, aventures et difficultés rencontrées au cours du développement/amélioration du cœur du processeur ou au cours de la réalisation du circuit.

Les perspectives de ces travaux vont dans une direction d'évolution et de validation avancée des architectures sur plusieurs plans :

– Prototypage sur des platformes multi-FPGA, car comme nous avons pu le voir les ressources matérielles sont telles que le plus grand FPGA Xilinx ne peut contenir l'architecture. Le but de ce prototypage est de fournir une validation fonctionnelle du futur circuit à fabriquer.

– Validation de l'architecture proposée avec d'autres algorithmes de traitement d'image. Cette validation permet d'obtenir plus d'information sur les améliorations à apporter à l'architecture. L'objectif est d'augmenter sa flexibilité afin d'élargir le domaine d'application.

– Augmentation du nombre de cœurs à plusieurs centaines afin d'obtenir des architectures plus performantes.

– Exploration de technologies plus intégrées telles que la 22nm et la 16nm pour plus de performances.

– Une exploration de la technologie 3D devrait s'avérer intéressante et prometteuse [83], notamment par l'isolation de la mémoire sur une couche du layout.

Annexe A

Cette annexe illustre les détails des développements matériels et logiciels effectués dans le chapitre 3.

A.1 Ajout des instructions FSL

Afin d'ajouter les instructions FSL au processeur, deux aspects sont à prendre en compte. Premièrement l'aspect logiciel qui consiste à rajouter les fonctions des instructions pour que le compilateur compile correctement l'instruction sous son format 32 bits. Deuxièmement l'aspect matériel qui concerne les étapes du décodage de l'instruction jusqu'à l'écriture de la donnée sur le port FSL pour une instruction d'écriture ou dans la banque de registre pour une instruction de lecture.

A.1.1 Aspects logiciels

Les fonctions suivantes sont définies comme des macros en MicroBlaze assembleur dans le fichier d'entête *mb_interface.h*. Dans notre version du projet le fichier d'entête est nommé *sb_fsl.h*.

A.1.1.1 Instructions de type GET

Les instructions de type GET (fig. A.1) sont des instructions de lecture. Elles lisent la valeur sur le port FSL et la sauvegardent dans la banque de registres.

• *sb_bread_datafsl(val, id)* : permet de lire la valeur du port FSL sélectionné par id et de sauvegarder la valeur dans la variable val. Cette instruction est de type bloquant, c'est-à-dire que le processeur est dans un état bloquant tant que la donnée à lire sur le port FSL est non valide.

• *sb_bread_cntlfsl(val, id)* : cette instruction est la même que la précédente à la seule différence que la valeur à lire et à sauvegarder dans val est une donnée de contrôle.

• *sb_nbread_datafsl(val, id)* : permet de lire la valeur du port FSL sélectionné par id et de sauvegarder la valeur dans la variable val. Cette instruction est de type non

bloquant c'est-à-dire qu'une valeur est lue dans tous les cas. Si la donnée est disponible dans la FIFO, le bit de carry du registre msr est mis à 0 sinon il est mis à 1.

• *sb_nbread_cntlfsl(val, id)* : cette instruction est la même que la précédente à la seule différence que la valeur à lire et à sauvegarder dans val est une donnée de contrôle.

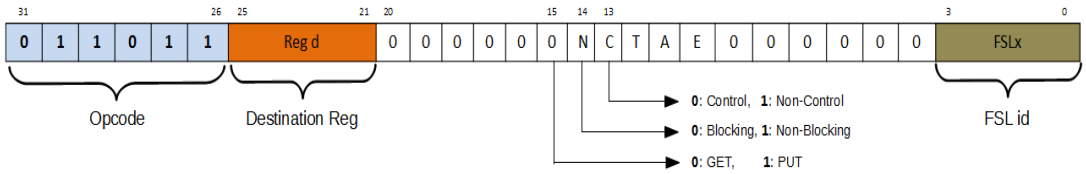


FIGURE A.1 – Registre de l'instruction GET

A.1.1.2 Instructions de type PUT

Les instructions de type PUT (fig. A.2) sont des instructions d'écriture. Elles permettent d'écrire une valeur définie dans le registre d'instruction sur le port FSL.

• *sb_bwrite_datafsl(val, id)* : permet d'écrire la valeur de la variable val sur le port FSL sélectionné par id. Cette instruction et de type bloquant c'est-à-dire que le processeur est dans un état bloquant tant que la FIFO est pleine.

• *sb_bwrite_cntlfsl(val, id)* : cette instruction est la même que la précédente à la seule différence que la valeur à écrire sur le port FSL est une donnée de contrôle.

• *sb_nbwrite_datafsl(val, id)* : permet d'écrire la valeur de la variable val sur le port sélectionné par id. Cette instruction et de type non bloquant c'est-à-dire que la valeur est écrite dans tous les cas. Si la FIFO est vide, le bit de carry du registre msr est mis à 0. Sinon il est mis à 1.

• *sb_nbwrite_cntlfsl(val, id)* : cette instruction est la même que la précédente à la seule différence que la valeur à écrire sur le port FSL est une donnée de contrôle.

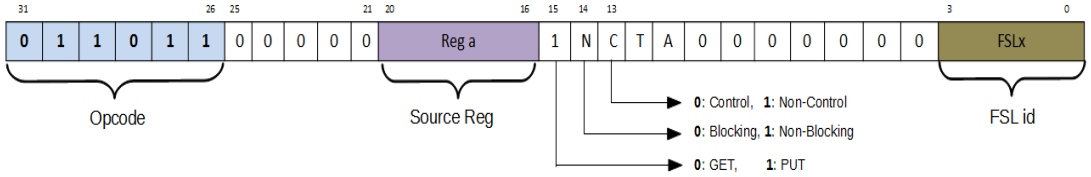


FIGURE A.2 – Registre de l'instruction PUT

A.1.1.3 Version dynamique des instructions FSL

Dans un programme où le port FSL n'est pas défini à priori, l'usage d'instructions FSL dites dynamiques s'impose (par exemple lorsque le numéro du port FSL choisi pour la communication dépend du traitement réalisé). A l'image de ce qui est développé dans le MicroBlaze, ces instructions ont le même comportement que les huit instructions détaillées précédemment. La seule différence est que le port FSL est adressé en mode indirect. L'instruction FSL contient alors l'adresse du registre contenant l'identificateur du port FSL à utiliser.

Dans les instructions de type GET de type dynamique (fig. A.3), le port FSL duquel on va lire la donnée est sélectionné par les 4 bits les moins significatifs du registre sélectionné par le registre "Reg b".

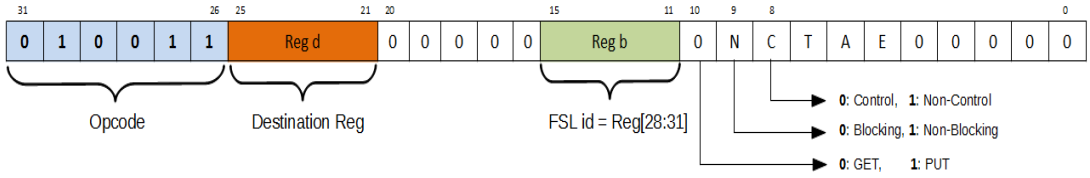


FIGURE A.3 – Registre de l'instruction GET dynamique

Dans les instructions de type PUT de type dynamique (fig. A.4), le port FSL dans lequel la donnée va être écrite est sélectionné par les 4 bits les moins significatifs du registre sélectionné par le registre "Reg b".

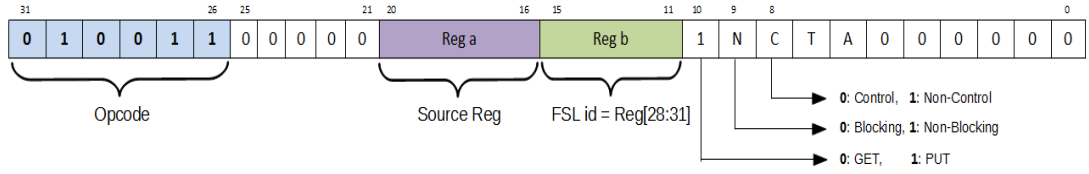


FIGURE A.4 – Registre de l'instruction PUT dynamique

A.1.2 Aspects matériels

Dans cette partie, les modifications matérielles effectuées pour implanter les instructions FSL sont présentées.

A.1.2.1 Modification du pipeline

Lorsque l'étage "recherche d'instruction" récupère une instruction FSL, il la transmet à l'étage suivant c'est-à-dire, l'étage de "décodage d'instructions". Dans cet étage,

il fallait concevoir la partie de décodage qui permet de générer les signaux nécessaires à l'exécution de l'instruction FSL (fig. A.5). Le signal gp_control est un signal qui permet d'indiquer si l'instruction en cours est de type PUT, GET ou GP_NOP. Le signal *fsl_control* contient quatre sous signaux permettant d'identifier le type d'instruction (GET ou PUT, Control ou non control, bloquant ou non bloquant) ainsi que le *id* permettant de sélectionner le port FSL. Au coup d'horloge suivant, ces signaux sont transmis à l'étage "d'exécution", dans lequel la donnée à écrire sur le port FSL (*fsl_data*) est récupérée si l'instruction en cours est de type PUT. Dans cet étage, le registre MSR est également mis à jour à partir des signaux *fsl_m_full*, *fsl_s_control* et *fsl_s_exists*. Au coup d'horloge suivant, ces signaux sont transmis à l'étage suivant "accès mémoire". Dans cet étage, le bloc "FSL Access" permet de sélectionner le port dans lequel la donnée va être écrite si c'est une instruction de type PUT, ou lue si c'est une instruction de type GET. Dans le cas d'une instruction de type GET, la donnée est récupérée à partir du port correspondant et transmise au coup d'horloge suivant à l'étage "retour écriture" qui s'occupe d'écrire la donnée dans la banque de registres se trouvant à l'étage "décodage d'instruction".

Afin d'augmenter le nombre de ports de communication et rendre le système beaucoup plus flexible, le bloc FSL Access est constitué de 2 multiplexeurs et 2 démultiplexeurs pilotés par le signal fsl_id (fig. A.5).

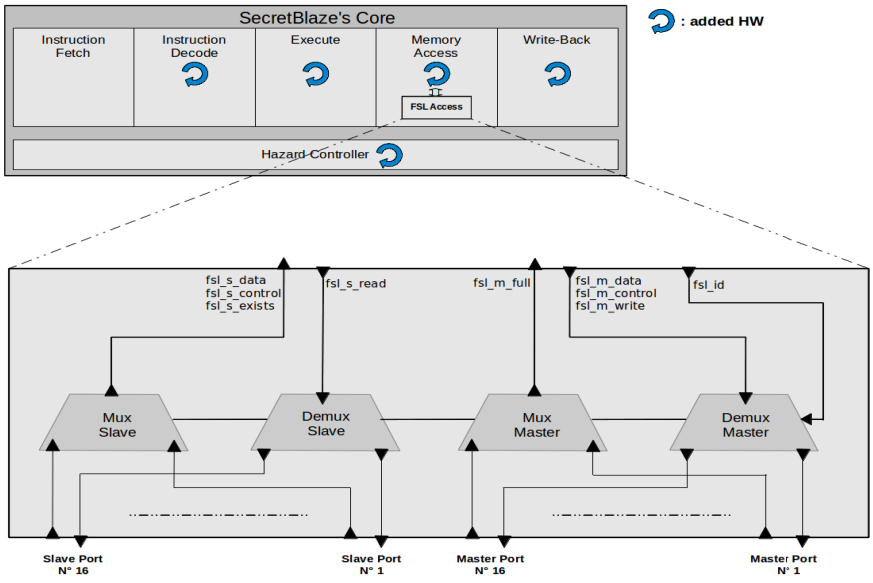


FIGURE A.5 – Modification du pipeline

A.1.2.2 Blocage du processeur

Dans les instructions de type bloquant, l'idée consiste à extraire du cœur du processeur les signaux indiquant qu'une instruction bloquante est en cours d'exécution

(fig. A.6). Ces signaux sont utilisés afin d'établir les conditions nécessaires au blocage : le résultat est injecté à l'entrée *halt-core* du cœur du processeur déclenchant ainsi un blocage si nécessaire. De même pour la mémoire qui utilise également le signal *halt_core* afin de bloquer le processeur. Par exemple, lorsqu'une donnée est non disponible (échec sur la mémoire cache), le processeur est bloqué le temps nécessaire afin que le cache soit mis à jour.

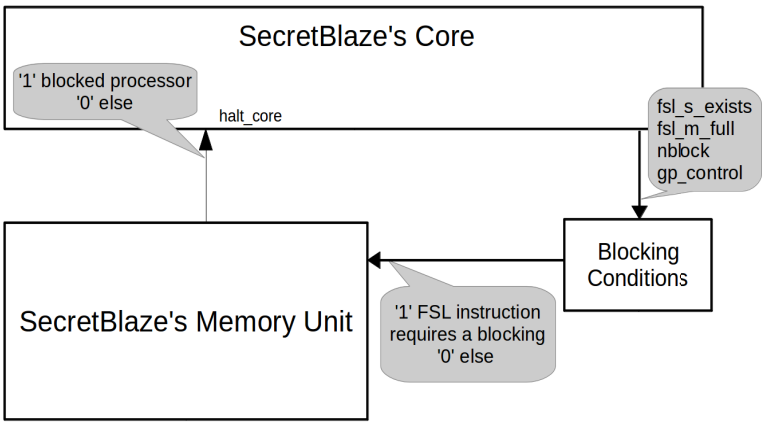


FIGURE A.6 – Architecture permettant le blocage du processeur

A.1.3 Vérification et validation de l'ajout des instructions FSL

Dans cette partie, nous allons vérifier le bon fonctionnement des instructions FSL. La méthodologie utilisée consiste, dans un premier temps, à faire une pré-vérification fonctionnelle avec un seul processeur qui est rapide est simple à mettre en œuvre, puis une validation avec deux processeurs.

A.1.3.1 Pré-vérification fonctionnelle

Le principe est le suivant :
- Connecter les 16 ports maîtres avec les 16 ports esclaves correspondant (fig. A.7)
- Réaliser une écriture sur un port master
- Lire la donnée sur le port esclave correspondant.

Les deux instructions suivantes montrent un exemple de programme de test :

```
sb_bwrite_datafsl(0x12345678, 7);
sb_bread_datafsl(val, 7);
```

La première instruction permet d'écrire la valeur 0x12345678 sur le port numéro 7. La deuxième instruction permet de lire la donnée sur le port 7 et de la sauvegarder dans la variable *val*.

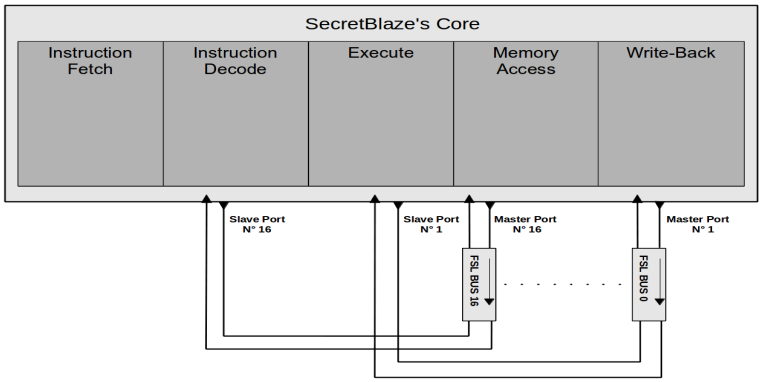


FIGURE A.7 – Architecture pour le test des instructions FSL

A.1.3.2 Validation sur un réseau de deux SecretBlaze

Afin de valider complètement la communication via FSL, une architecture composée de deux processeurs communicants via des liens FSL est proposée. Celle-ci permet de vérifier la connexion inter-processeurs. La figure A.8 montre l'architecture mise en place :

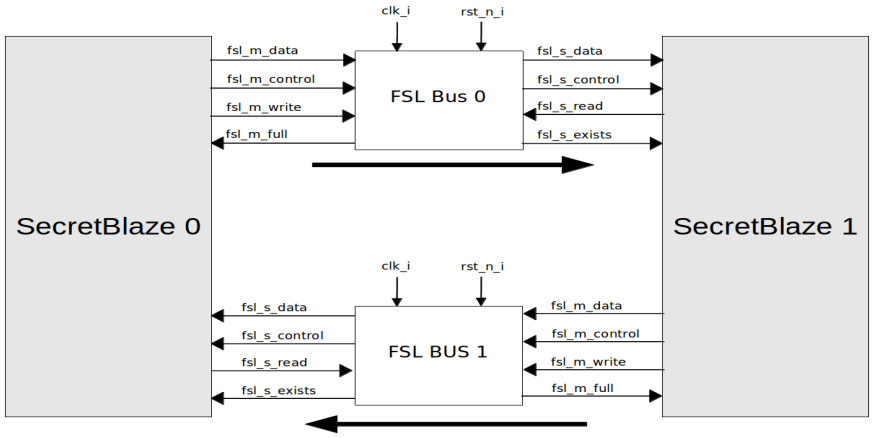


FIGURE A.8 – Architecture de deux processeurs communicants via des liens FSL

Le processeur SecretBlaze 0 envoie les données via le FSL Bus 0. Au coup d'horloge suivant les données arrivent au processeur SecretBlaze 1 qui lui va lire les données sur son port esclave. Ensuite, il réécrit les données sur son port maître via le FSL Bus 1. Les données sont alors transmises au processeur SecretBlaze 0. Au coup d'horloge suivant, les données arrivent au processeur SecretBlaze 0, ce dernier va les lire sur son port esclave.

A.2 Intégration de la FPU dans le système

La figure A.9 montre l'architecture mise en œuvre. Cinq ports sont connectés à la FPU via les bus FSL. Le bus FSL est implanté comme une FIFO pour l'envoi de données. Les fonctions suivantes (écrites en langue C) permettent au programmeur d'accéder aux différentes instructions FSL.

Dans le SecretBlaze, l'instruction PUT est utilisé pour écrire les données dans le bus FSL comme suit :

```
sb_bwrite_datafsl(operandA, 0); //write the opA on port 0
sb_bwrite_datafsl(operandB, 1); //write the opB on port 1
sb_bwrite_datafsl(oper_rm, 2);  //write the oper & rm on port 2
```

Les ports de l'opérande A et de l'opérande B de la FPU reçoivent les données des ports master 0 et 1 respectivement. Les ports *opération* et *mode d'arrondi* reçoivent les données du port master 2. Le module *FPU FSM busy* est une machine d'état qui gère le signal *start* en fonction de l'état de la FPU (occupé ou prêt à faire un autre traitement). Lorsque la FPU est prête, le signal *reday* est fixé à 1 ainsi que les signaux *write* et *read* des bus FSL. L'opérande A, l'opérande B, l'opération et le mode d'arrondi sont envoyés (popped) à partir des bus FSL pour faire le prochain calcul. Dans le même temps, le résultat (*output* et *status*) sont envoyés dans les bus FSL.

Dans le SecretBlaze, l'instruction GET est utilisée pour lire les données à partir du bus FSL comme suit :

```
sb_bread_datafsl(Output, 0); //read the output from port 0
sb_bread_datafsl(Status, 1); //read the status from port 1
```

D'un point de vue performances, une instruction FSL s'exécute en 1 cycle. Pour effectuer un calcul avec FPU, 5 instructions sont nécessaires. Le temps de calcul FPU est égal à 7 cycles pour une addition ou une soustraction, 12 cycles pour une multiplication et 35 cycles pour une division ou une racine carrée. Le temps total d'exécution de ces opérations peut être exprimée comme suit :

$$T_{total} = T_{fpu\_computing} + T_{fsl\_instructions} \tag{A.1}$$

$$T_{total} = T_{fpu\_computing} + 5cycles \tag{A.2}$$

$T_{total(add)}$ = 12 cycles
$T_{total(sub)}$ = 12 cycles
$T_{total(mult)}$ = 17 cycles
$T_{total(div)}$ = 40 cycles
$T_{total(suq)}$ = 40 cycles

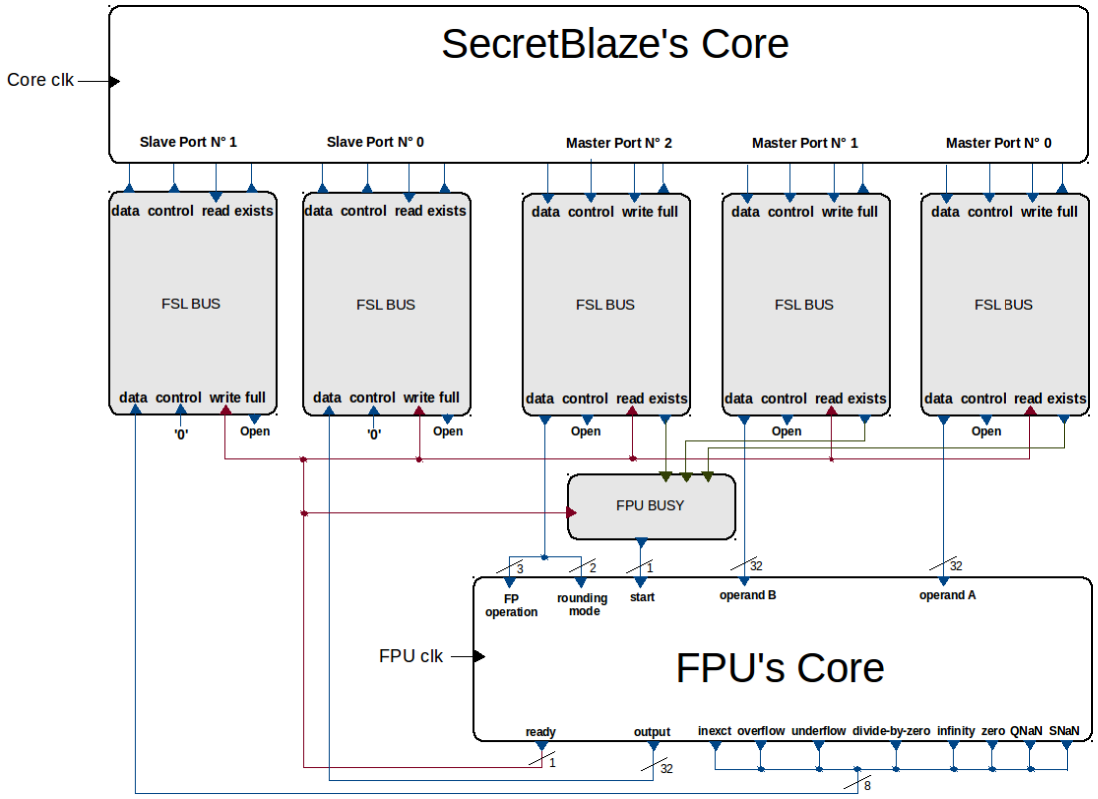


FIGURE A.9 – Vue d'ensemble sur l'architecture proposée

A.2.1 Optimisation à l'aide des configurations de bus FSL

Afin d'améliorer les performances et garder le processeur libre le plus longtemps possible lorsque la FPU est en cours de traitement, nous proposons deux solutions : la première ("mode asynchrone") améliore le temps de calcul de la FPU et la deuxième ("profondeur de la FIFO") permet de garder le processeur libre (les deux solutions utilisent la configuration des bus FSL) :

• **Mode asynchrone** La fréquence de la FPU peut être augmentée afin de réduire la latence en utilisant le mode asynchrone dans la configuration des bus FSL. Avec une fréquence d'horloge de la FPU plus rapide que la fréquence d'horloge du processeur, la performance de la FPU peut être considérablement améliorée. La fréquence maximale de fonctionnement du processeur utilisé (SecretBlaze) est évaluée à $172MHz$ sur le Virtex-6. La fréquence maximale de fonctionnement de la FPU choisi est la même que celle du processeur ($173MHz$). Dans ce cas, cette solution ne peut pas avoir un avantage pour notre conception, mais, ça peut être très intéressant avec une autre FPU plus rapide.

• **Profondeur de la FIFO** L'optimisation des performances peut être faite en réglant la profondeur de la FIFO du bus FSL en utilisant l'expression suivante :

$$FIFO\_SISE = B - B \times \frac{FRD}{FWR \times IDLE\_CYCLE\_RD} \tag{A.3}$$

Avec B la taille du *burst* d'écriture, FRD l'horloge d'écriture, FWR l'horloge d'écriture et $IDLE\_CYCLE\_RD$ cycle d'horloge idle pour la lecture.

Cette expression permet aux utilisateurs de configurer la profondeur de la FIFO en fonction de leurs applications. La profondeur de la FIFO du bus FSL respecte : $1 \leq profondeur \leq 8192$. Cette optimisation peut être utilisée pour réduire la latence dans le cas d'un calcul complexe. Le temps global ne change pas mais ça permet au processeur de mieux gérer plus efficacement le temps "libre" (voir temps libre disponible sur la figure A.11, et non disponible sur la figure A.10).

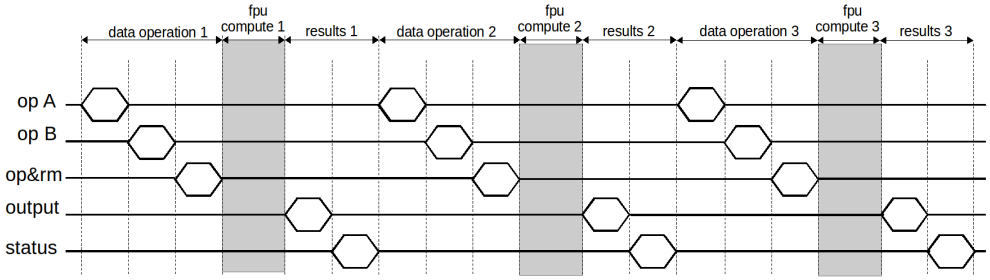


FIGURE A.10 – Chronogramme avec une FIFO de profondeur égale à 1

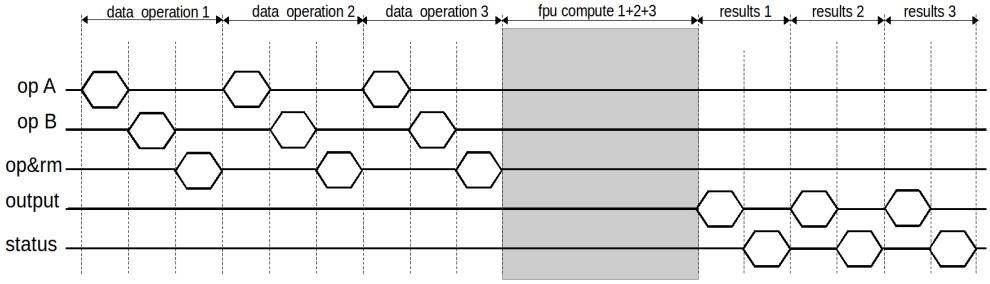


FIGURE A.11 – Chronogramme avec une FIFO de profondeur supérieur à 3

A.3 Le module JTAG esclave

Le module JTAG esclave présenté sur la figure. A.12 est constitué de trois sous modules : Le contrôleur (*JTAG Tap controller*), Le registre d'instructions (*JTAG instruction Register*) et les registres de données (*JTAG Data Registers*). Le contrôleur interprète les signaux présents sur le bus JTAG et génère les signaux de contrôle en direction des autres modules. Les données entrantes via le port *tdi* son acheminées sur chaque module contenant des registres. Un multiplexeur route les données sortantes issues de chaque module contenant des registres vers la sortie *tdo*.

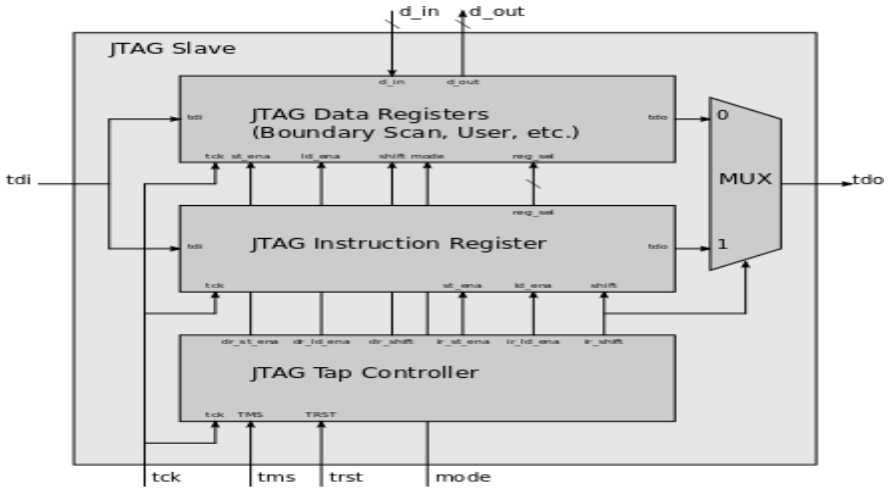


FIGURE A.12 – Le module JTAG esclave

A.3.1 Gestion des états (JTAG Tap Controller)

Le contrôleur intègre une machine d'état (fig. A.13) synchrone sur l'horloge *tck*. Elle comporte deux entrées : *tms* et *trst*. L'entrée optionnelle *trst* permet de ramener le contrôleur dans l'état *test_logic_reset*. Dans le cas où cette entrée n'est pas retenue, le contrôleur peut être ramené dans ce même état en fixant l'entrée *tms* à un pendant au moins cinq périodes d'horloge. Le contrôleur pilote six sorties permettant de contrôler les modules de registres.

A.3.2 Registre d'instructions (JTAG Intruction Register)

Le registre d'instruction (fig. A.14) est un registre à décalage contenant au moins deux cellules. Il est chargé avec *tdi* et déchargé sur *tdo*. Avant le décalage, il est chargé avec le mot présent sur l'entrée *d_in*. Ce mot est de longueur égale à la taille du registre. Seuls les deux bits les moins significatifs sont fixés et contiennent la valeur 0x02. A l'issue du décalage, le contenu du registre est présenté sur *d_out* de manière

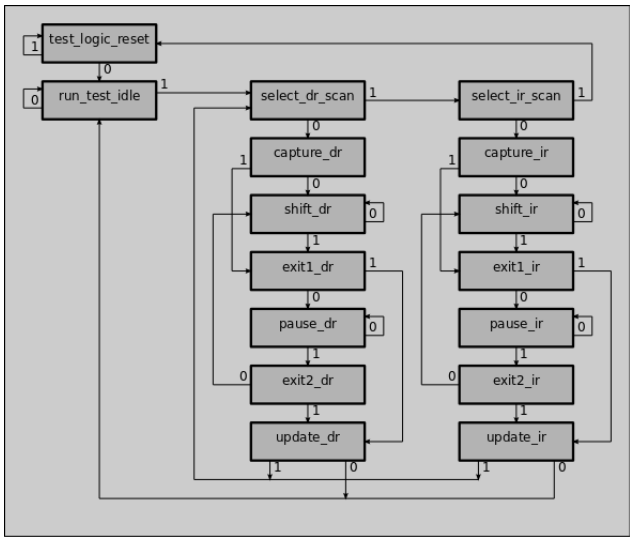


FIGURE A.13 – Graphe des états : l'évolution (synchrone) dépend de l'unique variable d'entrée TMS

stable jusqu'à l'issue du prochain décalage. Le signal *d_out* est décodé afin de générer un signal utilisé pour sélectionner l'un des registres de données.

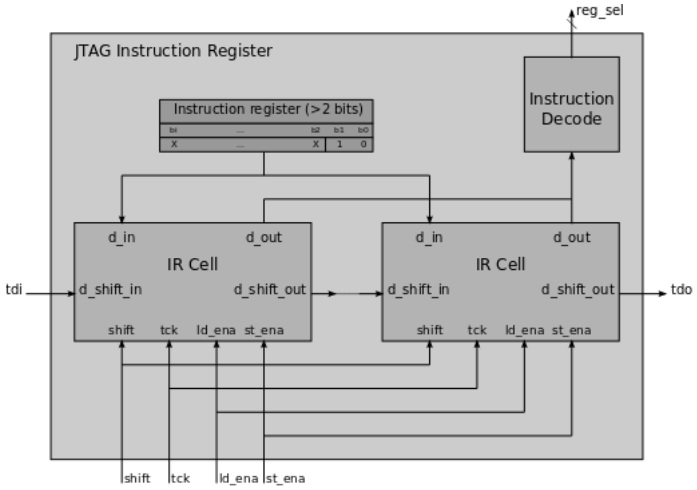


FIGURE A.14 – Architecture du registre d'instructions

A.3.3 Registres de données (JTAG Data Registers)

Le registre de données (fig. A.15) est constitué d'un module de décodage (Decode), d'un registre d'identification (DI Register), d'un registre de bypass (BP Register) et d'un registre de boundary scan (BS Register). Un multiplexeur route les données sor-

tantes issues de chaque module contenant des registres vers la sortie *tdo*.

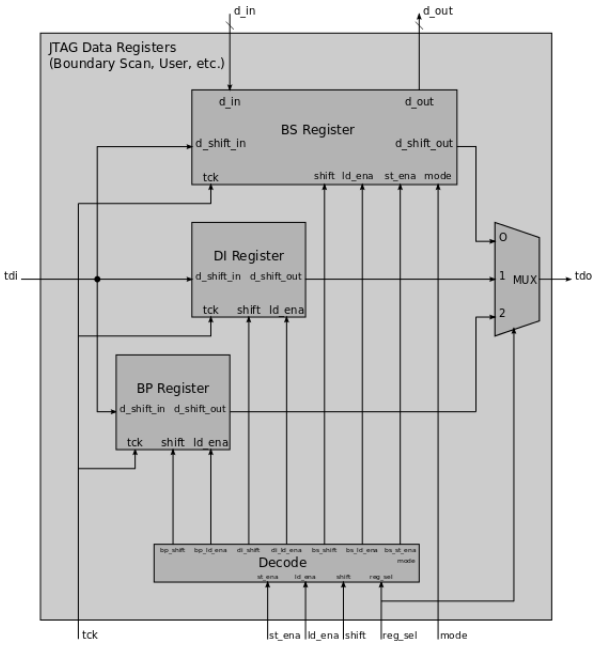


FIGURE A.15 – Architecture du module de registres de données

A.3.4 Interface USB/JTAG

Le module FT2232H de la figure A.16 est proposé par *FTDI Chip* [84]. Il permet de réaliser l'interfaçage USB/JTAG. Il est proposé avec des drivers fonctionnant sous environnement Windows (32 et 64 bits) ainsi que sous environnement Linux. L'accès au driver est possible à travers une API.

FIGURE A.16 – Le module FT2232H [84]

A.4 Exemples de pseudo-codes de routage pour des topologies grille et tore

La mise on œuvre des topologies est basée sur des comparaisons. C'est-à-dire que chaque nœud possède un ID sous la forme XY. La valeur X permet de situer le nœud sur l'axe des X et la valeur Y permet de situe le nœud sur l'axe des Y. Donc, un balayage sur l'axe Y est effectué, suivi d'un balayage sur l'axe Y.

Exemple pour une topologie grille : le pseudo-code de routage suivant montre un exemple de mise en œuvre pour un une architecture a 16 nœuds (4x4) dans une topologie grille :

```
Si dst_x < local_x  alors
  Channel_out = "00010". (le routage à gauche)
Sinon si dst_x > local_x alors
  Channel_out = "00100" (le routage à droite)
Sinon si dst_y < local_y alors
  Channel_out = "01000" (le routage vers le bas)
Sinon si dst_y > local_y  alors
  Channel_out = "10000" (le routage vers le haut)
Sinon "00001".
Fin si
```

Exemple pour une topologie tore : le pseudo-code de routage suivant montre un exemple de mise en œuvre pour un une architecture a 16 nœuds (4x4) dans une topologie tore :

```
Si (dst_x < local_x et (local_x -dst_x)<=2) ou
   (dst_x > local_x et (dst_x - local_x) > 2) alors
   Channel_ out = "00010" (le routage à gauche)
Sinon si (dst_x > local_x et (dst_x - local_x)<=2) ou
         (dst_x < local_x et (local_x -dst_x) > 2) alors
   Channel_ out = "00100" (le routage à droite)
Sinon si (dst_y < local_y et (local_y - dst_y)<=2) ou
         (dst_y > local_y et (dst_y -local_y) > 2) alors
   Channel_ out = "01000" (le routage vers le bas)
Sinon si (dst_y > local_y et (dst_y - local_y)<=2) ou
         (dst_y < local_y et (local_y - dst_y)>2 alors
   Channel_ out = "10000" (le routage vers le haut)
Sinon "00001".
Fin si
Fin si
```

Avec : local_x : nœud local sur l'axe des X. local_y : nœud local sur l'axe des Y. dst_x : nœud destination sur l'axe des X. dst_y : nœud destination sur l'axe des Y.

A.5 Exemples de pseudo-codes des fonctions logiciels du DMA-Router

L'exemple suivant montre le pseudo-code d'une configuration en émission du DMA-Router :

```
int dma_snd_config (unsigned int channel,
                    unsigned int destination_node_id,
            unsigned int message_id,
                    unsigned int message_lenght,
            unsigned int base_address)
{
  register unsigned int op_code;
  message_id = (message_id<<2) | channel;
  op_code = SET_SND_CONFIG | channel;
  sb_bwrite_datafsl(op_code, 0);
  sb_bwrite_datafsl(message_id, 0);
  sb_bwrite_datafsl(destination_node_id, 0);
  sb_bwrite_datafsl(message_lenght, 0);
  sb_bwrite_datafsl(base_address, 0);
}
```

L'exemple suivant montre le pseudo-code d'une configuration en réception du DMA-Router :

```
int dma_rcv_config (unsigned int channel,
            unsigned int source_node_id,
            unsigned int message_id,
            unsigned int message_lenght,
            unsigned int base_address)
{
  register unsigned int op_code;
  message_id = (message_id<<2) | channel;
  op_code = SET_RCV_CONFIG | channel;
  sb_bwrite_datafsl(op_code, 0);
  sb_bwrite_datafsl(message_id, 0);
  sb_bwrite_datafsl(source_node_id, 0);
  sb_bwrite_datafsl(message_lenght, 0);
  sb_bwrite_datafsl(base_address, 0);
}
```

Annexe B

Cette annexe illustre les détails des développements matériels et logiciels effectués dans le chapitre 4.

B.1 Prototypage du premier RUN sur FPGA

La figure B.1 montre la carte de développement (Spartan-6 LX45) et le module JTAG (FT2232H) utilisés dans le système de prototypage.

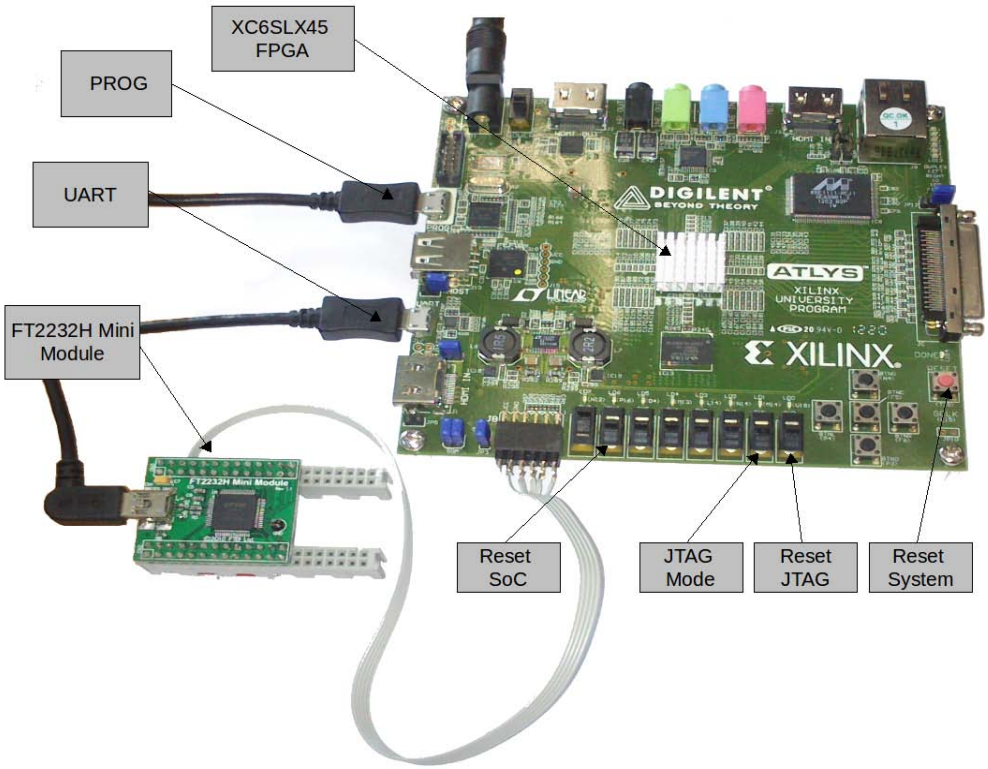


FIGURE B.1 – Prototypage du premier RUN

B.2 Simulation d'un programme "hello world"

Le chronogramme présenté par la figure B.2 montre le résultat de simulation d'une exécution de programme classique appelé "hello world". Deux phases peuvent être observées dans cette simulation : la première phase est la programmation de la mémoire processeur, cette phase commence du premier marqueur rouge au marqueur bleu et elle dure $3,55ms$. Une fois la programmation de la mémoire terminée, le processeur commence l'exécution du programme (la deuxième phase) lorsque le signal de réinitialisation (Reset) est mis à 1 (le deuxième marqueur rouge). Comme on peut le voir sur la la figure B.2, le signal de sortie du contrôleur UART commence le transfert des données peu après le commencement de l'exécution.

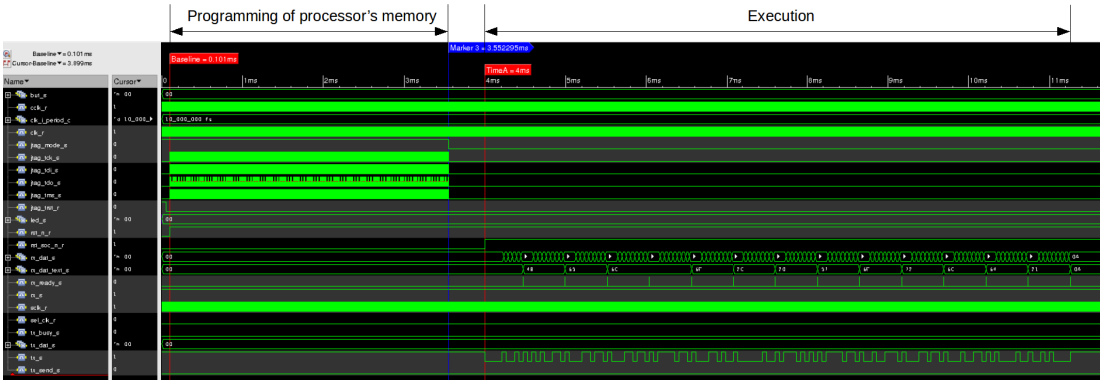


FIGURE B.2 – Simulation d'un programme "hello world"

Bibliographie

[1] L. Siéler. *Méthode de prototypage rapide par réseau de processeurs homogènes communicants pour le traitement d'images sur SoPC*. PhD thesis, Universite Blaise Pascal - Clermont II, Dec 2011.

[2] M. Bramberger, J. Brunner, B. Rinner, and H. Schwabach. Real-time video analysis on an embedded smart camera for traffic surveillance. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 174–181, May 2004.

[3] D. G. A. Rowe, A.G. Goode, and I. Nourbakhsh. Cmucam3 : An open programmable embedded vision sensor. Technical report, 2007.

[4] S. Hengstler, D. Prashanth, Sufen Fong, and H. Aghajan. Mesheye : A hybrid-resolution smart camera mote for applications in distributed intelligent surveillance. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 360–369, April 2007.

[5] Pierre Chalimbaud and François Berry. Embedded active vision system based on an fpga architecture. *EURASIP J. Embedded Syst.*, 2007(1) :26–26, January 2007.

[6] R Mosqueron, J Dubois, and M Paindavoine. High-speed smart camera with high resolution. *EURASIP Journal on Embedded Systems*, 2007(1) :024163, 2007.

[7] R. Kleihorst, A. Abbo, B. Schueler, and A. Danilin. Camera mote with a high-performance parallel processor for real-time frame-based video processing. In *Advanced Video and Signal Based Surveillance, 2007. AVSS 2007. IEEE Conference on*, pages 69–74, Sept 2007.

[8] Merwan Birem and François Berry. Dreamcam : A modular fpga-based smart camera architecture. *Journal of Systems Architecture*, 60(6) :519–527, 2014.

[9] DreamCam. *http ://dream.univ-bpclermont.fr/index.php/plateformemenu/dreamcam*.

[10] Cyclone3. *http ://www.altera.com/devices/fpga/cyclone3/cy3-index.jsp*.

[11] Altera. *http ://www.altera.com/*.

[12] M. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9) :948–960, Sept 1972.

[13] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38 :114–117, Apr 1965.

[14] Carlo Kopp. Moore's law and its implications for information warfare. In *The 3rd International AOC EW Conference (Invited Paper)*, Jan 2002.

[15] U. Meyer-Bäse. The use of complex algorithm in the realization of universal samplingreceiver using fpgas (in german). volume 10, page 215, mar 1995.

[16] Rainer Domer, Andreas Gerstlauer, Junyu Peng, Dongwan Shin, Lukai Cai, Haobo Yu, Samar Abdi, and DanielD Gajski. System-on-chip environment : A specc-based framework for heterogeneous mpsoc design. *EURASIP Journal on Embedded Systems*, 2008(1) :647953, 2008.

[17] Altera. *AN 311 : Standard Cell ASIC to FPGA Design Methodology and Guidelines*, 2009.

[18] Ian Kuon and J. Rose. Measuring the gap between fpgas and asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2) :203–215, Feb 2007.

[19] Paul Metzgen. A high performance 32-bit alu for programmable logic. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, pages 61–70, New York, NY, USA, 2004. ACM.

[20] EETimes. *http ://www.eetimes.com/.*

[21] J. Serot, F. Berry, and S. Ahmed. Caph : A language for implementing stream-processing applications on fpgas. pages 130–137, 2011.

[22] Sidi Ahmed Mahmoudi, Sébastien Frémal, Michel Bagein, and Pierre Manneback. Calcul intensif sur gpu : exemples en traitement d'images, en bioinformatique et en télécommunication. In *CIAE 2011 : Colloque d'informatique, automatique et électronique*, 2011.

[23] Altera. *Nios II Processor Reference*, 2014.

[24] Xilinx. *MicroBlaze Processor Reference Guide Embedded Development Kit EDK 12.2*, 2010.

[25] L. Damez. *Approche multi-processeurs homogenes sur System-On-Chip pour le traitement d'image.* PhD thesis, Universite Blaise Pascal - Clermont II, dec 2009.

[26] Claudio Brunelli, Fabio Campi, Claudio Mucci, Davide Rossi, Tapani Ahonen, Juha Kylliäinen, Fabio Garzia, and Jari Nurmi. Design space exploration of an open-source, ip-reusable, scalable floating-point engine for embedded applications. *Journal of Systems Architecture*, 54(12) :1143–1154, 2008.

[27] H. Chenini. *A Rapid design methodology for generating of parallel image processing applications and parallel architectures for smart camera.* PhD thesis, Universite Blaise Pascal - Clermont II, may 2014.

[28] J. P. Dérutin, B. Besserer, and J. Gallice. A parallel vision : Transvision. In *In Proceedings of CAMP-91, Computer Architectures for Machine Perception*, pages 241–251, 1991.

[29] D. Houzet. *Conception et étude de l'architecture parallèle de traitement d'image GFLOPS.* PhD thesis, INP Toulouse, Jan 1992.

[30] J. Sérot, G. Quenot, and B. Zavidovique. Functionnal programming on a dataflow architecture : applications in real time image processing. *Machine Vision and Applications*, pages 44–56, Jul 1993.

[31] T.A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins. Topology adaptive network-on-chip design and implementation. *Computers and Digital Techniques, IEE Proceedings* -, 152(4) :467–472, July 2005.

[32] Manuel Saldaña, Lesley Shannon, and Paul Chow. The routability of multiprocessor network topologies in fpgas. In *Proceedings of the 2006 International Workshop on System-level Interconnect Prediction*, SLIP '06, pages 49–56, New York, NY, USA, 2006. ACM.

[33] Brian N. Bershad and Matthew J. Zekauskas. Midway : Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical report, 1991.

[34] Kalray. *http ://www.kalray.eu/.*

[35] Adapteva. *http ://www.adapteva.com/.*

[36] M.B. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, and J. Kim. Evaluation of the raw microprocessor : an exposed-wire-delay architecture for ilp and streams. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 2–13, June 2004.

[37] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor : A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88–598, Feb 2008.

[38] S.R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *Solid-State Circuits, IEEE Journal of*, 43(1) :29–41, Jan 2008.

[39] Zhiyi Yu, M.J. Meeuwsen, R.W. Apperson, O. Sattari, M. Lai, J.W. Webb, E.W. Work, D. Truong, T. Mohsenin, and B.M. Baas. Asap : An asynchronous array of simple processors. *Solid-State Circuits, IEEE Journal of*, 43(3) :695–705, March 2008.

[40] D.N. Truong, W.H. Cheng, T. Mohsenin, Zhiyi Yu, A.T. Jacobson, G. Landge, M.J. Meeuwsen, C. Watnik, A.T. Tran, Zhibin Xiao, E.W. Work, J.W. Webb, P.V. Mejia, and B.M. Baas. A 167-processor computational platform in 65 nm cmos. *Solid-State Circuits, IEEE Journal of*, 44(4) :1130–1144, April 2009.

[41] D. Truong, W. Cheng, T. Mohsenin, Zhiyi Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, Anh Tran, J. Webb, E. Work, Zhibin Xiao, and B. Baas.

A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 22–23, June 2008.

[42] Vipress. *http ://semiconductor.vipress.net/print.php ?J=tvyvajhc1709gmxj&T=2.*

[43] Tom's Hardware. *http ://www.tomshardware.fr/.*

[44] Plume. *https ://www.projet-plume.org/.*

[45] OpenRISC. *http ://opencores.org/or1k/OR1200_OpenRISC_Processor.*

[46] Plasma. *http ://opencores.org/project,plasma.*

[47] aeMB. *http ://opencores.org/project,aemb.*

[48] Aeroflex Gaisler. *GRLIB IP Core User's Manual*, 2010.

[49] OpenFire. *http ://opencores.org/project,openfire_core.*

[50] ADAC. *http ://www.lirmm.fr/ADAC/ ?page_id=462.*

[51] MB-LITE. *http ://opencores.org/project,mblite.*

[52] T. Kranenburg and R. van Leuken. Mb-lite : A robust, light-weight soft-core implementation of the microblaze architecture. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 997–1000, March 2010.

[53] L. Barthe, L.V. Cargnini, P. Benoit, and L. Torres. Optimizing an open-source processor for fpgas : A case study. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 551–556, Sept 2011.

[54] T. Kranenburg. Reference design of a portable and customizable microprocessor forrapid system prototyping. Master's thesis, 2009.

[55] OpenCores. *Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, 2010.

[56] Xilinx. *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11e)*, 2011.

[57] Rudolf Usselmann. *http ://opencores.org/project,fpu.*

[58] John Hauser. *http ://www.jhauser.us/arithmetic/SoftFloat.html.*

[59] Jidan Al-Eryani. *http ://opencores.org/project,fpu100.*

[60] Marcus Guillermo. *http ://opencores.org/project,fpuvhdl.*

[61] Lundgren David. *http ://opencores.org/project,fpu_double.*

[62] Analog Devices. *Inc.ADSP-218x DSP Instruction Set Reference*, 2004.

[63] Analog Devices. *http ://www.analog.com/en/index.html.*

[64] Origin. *http ://www.ritme.com/fr/product/origin/description.*

[65] IEEE. Ieee standard test access port and boundary-scan architecture. Technical report, 1993.

[66] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2) :62–76, Feb 1993.

[67] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *Computers, IEEE Transactions on*, C-36(5) :547–553, May 1987.

[68] DREAM. *http ://dream.univ-bpclermont.fr/index.php/softmenu/ip-library*.

[69] STMicroelectronics. *http ://www.st.com/web/en/home.html*.

[70] AMS. *https ://www.ams.com/eng*.

[71] TSMC. *http ://www.tsmc.com/english/default.htm*.

[72] CMP. *http ://cmp.imag.fr/*.

[73] OBS. *http ://www.obs-nancay.fr/*.

[74] CMC. *http ://www.cmc.ca/en.aspx*.

[75] MOSIS. *http ://www.mosis.com/*.

[76] VDEC. *http ://www.vdec.u-tokyo.ac.jp/English/index.html*.

[77] Med Aymen SIALA and Slim BEN SAOUD. A survey on existing mpsocs architectures. *International Journal of Computer Applications*, 19(3) :28–41, April 2011.

[78] Chris Harris and Mike Stephens. A combined corner and edge detector. In *In Proceeding of the 4th Alvey Vision Conference*, page 147–151, 1988.

[79] Radu Gabriel Danescu. Image processing–laboratory work 3 : The histogram of image intensity levels. In *http ://users.utcluj.ro/ rdanescu/*, 2014.

[80] Derek Bradley and Gerhard Roth. Adaptive thresholding using the integral image. *Journal of Graphics, GPU, and Game Tools*, 12(2) :13–21, 2007.

[81] Eric Royer. *Cartographie 3D et localisation par vision monoculaire pour la navigation autonome d'un robot mobile*. PhD thesis, Université Blaise Pascal - Clermont II, Sep 2006.

[82] proFPGA. *http ://www.prodesign-europe.com/proFPGA_Products.html*.

[83] O. Hammami, A. M'zah, M.H. Jabbar, and D. Houzet. 3d ic implementation for mpsoc architectures : Mesh and butterfly based noc. In *Quality Electronic Design (ASQED), 2012 4th Asia Symposium on*, pages 155–159, July 2012.

[84] FTDICHIP. *http ://www.ftdichip.com/*.